版权注意事项:
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

VISIT…

LANZAROTE Caliente.com

HZ BOOKS
华章IT

区块链技术丛书

以全方位视角，结合生活化的示例与图表生动讲解，从技术、应用到系统设计

涵盖区块链底层技术、典型业务场景设计、主流框架与应用，并手把手教你从零构建区块链系统（微链）



白话区块链

蒋勇 文延 嘉文 著

机械工业出版社
China Machine Press

内容简介

以全方位视角，结合生活化的示例与图表生动讲解，从技术、应用到系统设计。本书涵盖区块链底层技术、典型业务场景设计、主流框架与应用，并手把手教你从零构建区块链系统（微链）本书共 9 章，以下为涉猎的内容。

第 1 章 通过村民记账的场景巧妙串起区块链的技术思想、技术组成，并以比特币为例介绍了基础技术原理。

第 2 章 综合介绍了典型区块链应用场景与流程。

第 3 章 介绍了现代密码算法在区块链中的作用与场景。

第 4 章 介绍了主流网络共识算法及其社会学价值。

第 5 章 介绍了区块链的链内外互联扩展技术的思路与主流做法。

第 6 章 详细介绍了以太坊的技术结构以及智能合约开发。

第 7 章 详细介绍了超级账本项目以及 Fabric 的配置与使用。

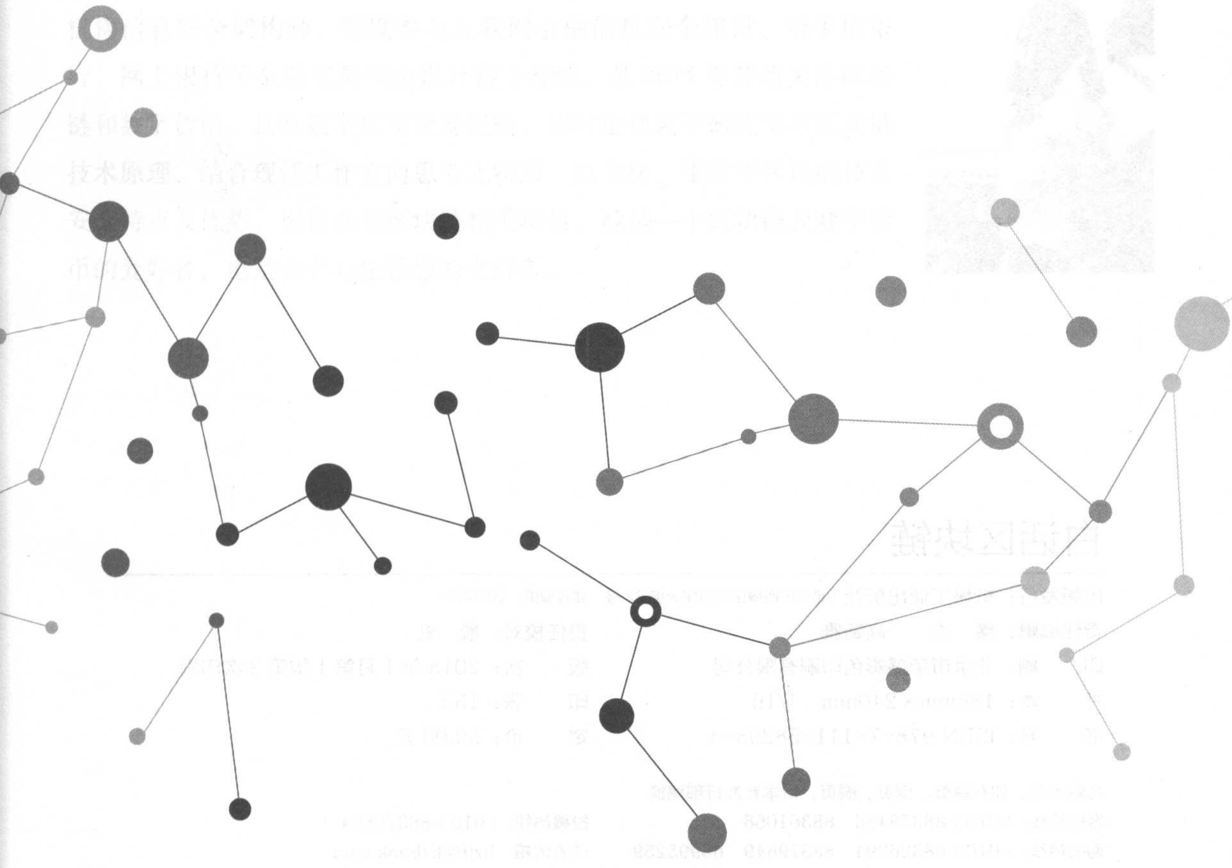
第 8 章 详细介绍了如何从零开始设计一个微型区块链系统（微链），加深理解并提升动手能力。

第 9 章 介绍了目前出现的各种区块链技术问题，多关于性能与安全。

区块链
技术丛书

白话区块链

蒋勇 文延 嘉文 著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

白话区块链 / 蒋勇，文延，嘉文著．—北京：机械工业出版社，2017.10（2018.1 重印）
（区块链技术丛书）

ISBN 978-7-111-58298-4

I. 白…　II. ①蒋…　②文…　③嘉…　III. 分布式计算机系统－研究　IV. TP338.8

中国版本图书馆 CIP 数据核字（2017）第 256191 号

白话区块链

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：缪　杰　　高婧雅　　责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2018 年 1 月第 1 版第 2 次印刷
开　　本：186mm×240mm　1/16　　印　　张：15.5
书　　号：ISBN 978-7-111-58298-4　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有・侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

Reviewer 技术审校



韩璐，毕业于北京工业大学计算机科学与技术专业，现任大型金融机构信息安全架构师，深度参与互联网金融信息安全建设，对手机银行、网上银行等金融交易安全设计富于经验。从 2014 年开始关注区块链和数字货币，具有数字货币交易经验，同时也热衷于研究学习区块链技术原理，结合现任工作方向思考比特币、以太坊、零币等区块链技术安全特点及优势，也曾参与区块链相关项目。她是一个区块链及数字货币的爱好者，也是去中心化思想的支持者。

前　言 *Preface*

为什么要写这本书

想要写一本综合介绍区块链的书，这个想法是从2016年年底开始有的。一直以来，关于这方面的资料比较少，能够找到的资料，或着眼于经济金融方面的发展远景，或着重介绍区块链的发展历史，或阐述纯技术化的内容，读来总是有一种意犹未尽的感觉。而身边的朋友或对区块链完全陌生，或是有很多误解，还有些朋友甚至简单地认为区块链就等于比特币。笔者也曾多次在一些类似读书会的场合对区块链进行较为通俗的介绍，然而很多感兴趣的朋友来自银行、投融资等行业，他们并非都有完备的计算机知识背景，当然也不乏一些希望从事区块链技术开发的程序员。然而即便是用了自认为很通俗的文字和语言来介绍，也难以在短短的一两个小时内讲清楚，对于各种名词术语、各种新鲜概念，每当他们希望我推荐一些资料的时候，我都很头疼。对于一个还没有广为人知的事物，大家的求知欲是很强烈的，并不满足于囫囵吞枣地了解概念，但也不喜欢去啃枯燥深入的技术文字，他们只是希望能有一个系统化的介绍，白话点的，通俗些的，能把每个点都讲到，把技术原理、应用场景、发展历史、当前现状等都贯穿起来。鉴于此，写这么一本书的想法就愈发强烈了。

我自2012年由比特币开始关注区块链技术，一直只在一个小范围的技术圈内进行讨论交流，每每为理解了一个技术概念而欣喜不已。区块链技术绝不仅仅代表一种数字货币，某种程度上，与其说是一门技术不如说是一类思想或者价值观。比特币把区块链技术带入了世人的眼中，以一种“货币”的身份降临，着实带来了不少的神秘感，其带来的理念为后来者所发扬光大，闪电网络、比特股、以太坊、超级账本等，不断冒出各种新的理念和产品，它们都是为了解决某一特定问题以及应用到更多领域而发展起来的。区块链技术的各种特点（分布式、可信任、不可篡改、智能合约等），在与传统技术领域结合的过程中，一定会显示出巨大的优势。事实上这两年区块链技术的发展可以说是势如破竹，相当迅猛，国内外都开始有大

量的机构或者企业投入研究，力图能够抓住这未来的一缕阳光。

这一切，都要从全面了解区块链开始。

本书将呈现给读者一个全方位的视角，从技术到应用以及未来展望，以通俗的语言阐述区块链的各个技术点，力求给读者一个通透的讲解，并希望能抛砖引玉，引导读者拓展出新颖而有价值的思路。

本书特色

从章节安排来说，本书从比特币开始，到区块链技术的骨骼（密码算法）和灵魂（共识算法），再到目前知名的系统，最后到从零开始构建一个微型区块链系统。读者的学习是一个由生到熟的渐进过程，对区块链完全陌生的读者，可以先从章节中的非专业技术部分读起，对于已经有一定基础的读者，可以从中挑选感兴趣的内容。

从内容安排来说，除了概念与原理的介绍之外，更多的是各种示例以及图表，以大量示例介绍比特币的源码编译、以太坊智能合约的开发部署、超级账本 Fabric 的配置使用、模拟比特币的微型区块链系统的设计实现等。阐述中会使用各种示意图，形象、直观地帮助读者理解各个概念和过程。

行文风格方面，力求白话通俗，避免枯燥感，使阅读体验更好。

读者对象

- 希望进行区块链开发的程序员。
- 希望投资或参与区块链项目的人员。
- 对区块链感兴趣的爱好者。

如何阅读本书

第 1 章　介绍区块链的技术组成，并以比特币为例介绍各种基础技术原理。

第 2 章　综合介绍目前的各种区块链应用，为后面的技术介绍铺垫场景。

第 3 章　介绍现代密码算法在区块链中的作用。

第 4 章　介绍各种网络共识算法。

第 5 章　介绍区块链的链内外互联扩展技术。

第 6 章　详细介绍以太坊的技术结构以及智能合约开发。

第 7 章　详细介绍超级账本项目以及 Fabric 的配置使用。

第 8 章　详细介绍如何从零开始设计一个微型区块链系统（简称微链）。

第 9 章　介绍目前出现的各种区块链技术问题。

勘误和支持

由于笔者水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果你有更多的宝贵意见，欢迎通过微信或邮件进行讨论。你可以通过微信 Cshen003、微博 @ 行者 C 神，或者发送邮件到邮箱 tnix_blockchain@outlook.com 联系到我，我会尽量给出满意的解答，期待能够得到你们的真挚反馈，在技术之路上互勉共进。

致谢

感谢我的作者伙伴——文延和嘉文，他们在工作之余，挤出宝贵的时间为本书贡献了他们对区块链技术的深入理解以及应用的展望分析，他们的专业和敬业令我感到钦佩。

感谢韩璐女士为本书做的审核工作，为书稿的内容质量付出了辛勤的劳动。

感谢比特币社区、以太坊社区、超级账本社区以及巴比特论坛各位技术专家，每次阅读他们的技术文章都让我有所收获，本书也多处引用了他们的观点和思想。

感谢中本聪，是他带来了区块链！

特别致谢

最后，感谢父母从小对我的培养，他们为我创造了良好的学习环境并培养了我爱好读书的习惯，这个习惯将伴随我终生并使我受益匪浅。因为工作和写书，牺牲了很多陪伴家人的时间，所以我更要感谢太太王晓英长期以来对我的默默支持，以及女儿 Cindy 对我工作的理解。

谨以此书献给我最亲爱的家人，多年以来帮助、支持我的朋友们，以及众多热爱区块链技术的朋友们！

蒋　勇

Contents 目　　录

技术审校

前言

第1章　初识区块链 ······ 1

1.1　例说区块链 ······ 1

1.1.1　从一本账本说起 ······ 1

1.1.2　区块链技术理念 ······ 3

1.1.3　一般工作流程 ······ 4

1.2　区块链技术栈 ······ 5

1.3　区块链分类与架构 ······ 10

1.3.1　区块链架构 ······ 10

1.3.2　区块链分类 ······ 13

1.4　一切源自比特币 ······ 15

1.4.1　比特币技术论文介绍 ······ 16

1.4.2　比特币核心程序：中本聪客户端 ······ 18

1.4.3　比特币的发行：挖矿 ······ 30

1.4.4　比特币钱包：核心钱包与轻钱包 ······ 35

1.4.5　比特币账户模型：UTXO ······ 39

1.4.6　动手编译比特币源码 ······ 41

1.5　区块链的技术意义 ······ 48

1.6　知识点导图 ······ 51

第2章　区块链应用发展 ······ 53

2.1　比特币及其朋友圈：加密数字货币 ······ 53

2.1.1 以太坊……54
2.1.2 比特币现金……56
2.1.3 莱特币……57
2.1.4 零币……57
2.1.5 数字货币发展总结……59
2.2 区块链扩展应用：智能合约……61
2.2.1 比特币中包含的合约思想……61
2.2.2 以太坊中图灵完备的合约支持……62
2.3 交易结算……62
2.3.1 银行结算清算……62
2.3.2 瑞波：开放支付网络……64
2.4 IPFS：星际文件系统……65
2.5 公证防伪溯源……66
2.6 供应链金融……70
2.7 区块链基础设施：可编程社会……74
2.8 链内资产与链外资产……76
2.9 知识点导图……77

第3章 区块链骨骼：密码算法……79

3.1 哈希算法……79
3.1.1 什么是哈希计算……79
3.1.2 哈希算法的种类……80
3.1.3 区块链中的哈希算法……81
3.2 公开密钥算法……83
3.2.1 两把钥匙：公钥和私钥……83
3.2.2 RSA 算法……84
3.2.3 椭圆曲线密码算法……85
3.3 编码 / 解码算法……86
3.3.1 Base64……87
3.3.2 Base58……88
3.3.3 Base58Check……89

3.4 应用场景 …… 90
3.5 知识点导图 …… 91

第4章 区块链灵魂：共识算法 …… 92

4.1 分布式系统的一致性 …… 92
4.1.1 一致性问题 …… 93
4.1.2 两个原理：FLP 与 CAP …… 94
4.1.3 拜占庭将军问题 …… 95
4.1.4 共识算法的目的 …… 96
4.2 Paxos 算法 …… 98
4.3 Raft 算法 …… 99
4.4 PBFT 算法 …… 101
4.5 工作量证明——PoW …… 102
4.6 股权权益证明——PoS …… 104
4.7 委托权益人证明机制——DPoS …… 104
4.8 共识算法的社会学探讨 …… 106
4.9 知识点导图 …… 107

第5章 区块链扩展：扩容、侧链和闪电网络 …… 108

5.1 比特币区块扩容 …… 108
5.2 侧链技术 …… 113
5.3 闪电网络的设计 …… 116
5.4 多链：区块链应用的扩展交互 …… 121
5.5 知识点导图 …… 122

第6章 区块链开发平台：以太坊 …… 123

6.1 项目介绍 …… 123
6.1.1 项目背景 …… 123
6.1.2 以太坊组成 …… 125
6.1.3 关键概念 …… 127
6.1.4 官方钱包使用 …… 143

6.2 以太坊应用 …… 151
6.2.1 测试链与私链 …… 151
6.2.2 编写一个代币合约 …… 158
6.3 知识点导图 …… 164

第7章 区块链开发平台：超级账本 …… 166
7.1 项目介绍 …… 166
7.1.1 项目背景 …… 166
7.1.2 项目组成 …… 167
7.2 Fabric 项目 …… 169
7.2.1 Fabric 基本运行分析 …… 169
7.2.2 Fabric 安装 …… 170
7.3 Fabric 示例 …… 173
7.3.1 部署准备 …… 173
7.3.2 启动 Fabric 网络 …… 178
7.3.3 Fabric 智能合约 …… 180
7.3.4 Fabric 部署总结 …… 187
7.4 知识点导图 …… 187

第8章 动手做个实验：搭建微链 …… 189
8.1 微链是什么 …… 189
8.2 开发环境准备 …… 190
8.3 设计一个简单的结构 …… 191
8.4 源码解析 …… 193
8.4.1 目录结构 …… 193
8.4.2 代码之旅 …… 194
8.5 微链实验的注意问题 …… 214
8.6 知识点导图 …… 214

第9章 潜在的问题 …… 216
9.1 两个哭泣的婴儿：软分叉与硬分叉 …… 217

9.2 达摩克利斯剑：51% 攻击 …… 220
9.3 简单的代价：轻钱包的易攻击性 …… 222
9.4 忘了保险箱密码：私钥丢失 …… 223
9.5 重放攻击：交易延展性 …… 225
9.6 代码漏洞：智能合约之殇 …… 227
9.6.1 说说 TheDAO 事件 …… 227
9.6.2 Parity 多重签名漏洞 …… 228
9.7 网络拥堵：大量交易的确认延迟 …… 229
9.8 容量贪吃蛇：不断增长的区块数据 …… 231
9.9 知识点导图 …… 232

后记 区块链与可编程社会 …… 234

第 1 章 Chapter 1

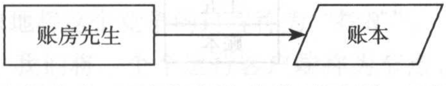
初识区块链

本章我们将从区块链的原理及分类、技术组成、技术特点等出发来初步介绍区块链的概念，并通过分析比特币的结构让大家对区块链有一个感性的认识。比特币作为区块链技术的第一个应用，它的原理设计影响深远。

1.1 例说区块链

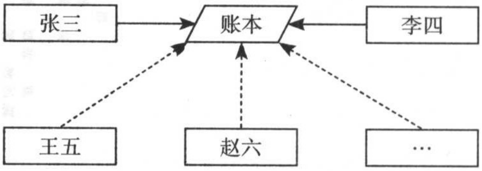
1.1.1 从一本账本说起

早些时候，农村一般都会有个账房先生，村里人出个工或者买卖些种子肥料等，都会依靠这个账房先生来记账，大部分情况下其他人也没有查账的习惯，那个账本基本就是这个账房先生保管着，到了年底，村长会根据账本余额购置些琐碎物件给村里人发发，一直以来也都是相安无事，谁也没有怀疑账本会有什么问题。账房先生因为承担着替大家记账的任务，因此不用出去干活出工，额外会有些补贴，仅此一点，倒也是让一些人羡慕不已。下图便是当时账本的记账权图示：



终于有一天，有个人无意中发现了账房先生的那本账。看了下账面，发现数字不对，最关键的是支出、收入、余额居然不能平衡。对不上，这可不行，立即报告给其他人，结果大家都不干了，这还得了。经过一番讨论，大家决定，轮流来记账，这个月张三，下个月李四，大家轮着来，防止账本被一个人拿在手里。于是，账本的记账权发生了如下图所

示的变化：



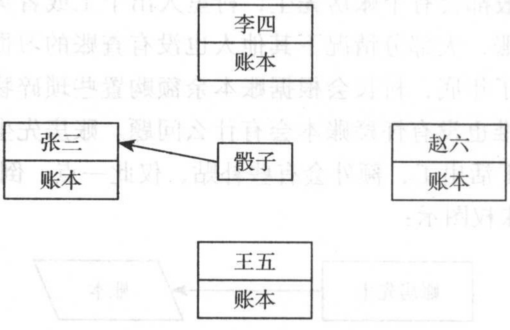
通过上图我们可以看到，村里的账本由大家轮流来保管记账了，一切又相安无事了，直到某一天，李四想要挪用村里的公款，可是他又怕这个事情被后来记账的人发现，怎么办呢？李四决定烧掉账本的一部分内容，这样别人就查不出来了，回头只要告诉大家这是不小心碰到蜡烛，别人也没什么办法。

果然，出了这个事情以后，大家也无可奈何。可是紧接着，赵六也说不小心碰到蜡烛了；王五说不小心掉水里；张三说被狗啃了……终于大家决定坐下来重新讨论这个问题。经过一番争论，大家决定启用一种新的记账方法：每个人都拥有一本自己的账本，任何一个人改动了账本都必须要告知所有其他人，其他人会在自己的账本上同样地记上一笔，如果有人发现新改动的账目不对，可以拒绝接受，到了最后，以大多数人都一致的账目表示为准。

果然，使用了这个办法后，很长一段时间内都没有发生过账本问题，即便是有人真的不小心损坏了一部分账本的内容，只要找到其他的人去重新复制一份来就行了。

然而，这种做法还是有问题，时间长了，有人就偷懒了，不愿意这么麻烦地记账，就希望别人记好账后，自己拿过来核对一下，没问题就直接抄一遍。这下记账记得最勤的人就有意见了。最终大家开会决定，每天早上掷骰子，根据点数决定谁来记当天的账，其他人只要核对一下，没问题就复制过来。

我们可以看到，在这个时候，账本的记账权变成了这样：



通过上图，我们可以看到，经历了几次风雨之后，大家终于还是决定共同来记账，这样是比较安全的做法，也不怕账本损坏丢失了。后来大家还决定，每天被掷到要记账的人，能获得一些奖励，从当天的记账总额中划出一定奖励的比例。

实际上，最后大家决定的做法，就是区块链中记账方法的雏形了，接下来我们就来了

解一下区块链的技术理念。

1.1.2 区块链技术理念

区块链在本质上就是一种记账方法，当然了，并不是通过人来记账的，而是通过一种软件，我们暂且简称为区块链客户端。以上面的例子来说，张三、李四、王五、赵六等人，就相当于一个个的区块链客户端软件，它们运行在不同的设备上，彼此之间独立工作。通常我们把运行中的客户端软件称为“节点”。这些节点运行后，彼此之间会认识一下。它们彼此之间是这样认识的：张三认识李四也认识王五，赵六联系到了张三，让张三把他认识的人的联系方式发给自己，这样赵六也认识了李四和王五，通过这样的方式，大家就形成了一张网，有什么事只要招呼一声，立马消息就会传遍整个网络节点。这种方式跟新闻转发差不多，不需要依靠某一个人，大家就能互通消息了，在区块链软件的结构中，这种互相通信的功能称为“网络路由”。

在这个网络中，每个节点都维护着自己的一个账本，账本中记录着网络中发生的一笔笔账务。具体是什么样的账务呢？这得看具体是什么样的功能网络。区块链技术属于一种技术方法，可以用来实现各种不同的业务功能，小到如上例中的日常记账，大到各种复杂的商业合约，等等，记录的数据也就不同了。网络中的节点是独立记账的，可是记账的内容要保持彼此一致。所用的方法就是设定一个游戏规则，通过这个规则选出一个记账的节点，就如上例中的掷骰子。在区块链系统中，这个所谓的“掷骰子”称为“共识算法”，就是一种大家都遵守的筛选方案，我们可以先这么简单地理解。选出一个节点后，则一段时间内的账务数据都以这个节点记录的为准，这个节点记录后会把数据广播出去，告诉其他的节点，其他节点只需要通过网络来接收新的数据，接收后各自根据自己现有的账本验证一下能不能接得上，有没有不匹配和不规范的，如果都符合要求，就存储到自己的账本中。

在有些系统中，会考虑到被骰子投中的节点的劳动付出，毕竟它要负责整理数据，验证数据，打包数据，还要再广而告之，这个活还是挺辛苦的。于是会设计一种激励机制，负责打包数据的那个节点可以获得系统的奖励，这个奖励类似于论坛积分，站在软件技术的角度，就是一个数据。这个数据可以视为奖金，有时候大家会很积极地去争取那个奖金，于是就希望骰子能投中自己，有些区块链系统在这个环节会设计出一种带有竞争的机制，让各个节点去抢，谁能抢到这个机会谁就能获得打包数据的权力并且同时获得这笔奖励，在这种情况下，我们会形象地将这个竞争的过程称为“挖矿”。

那么，话又说回来了，我们将一个个运行客户端称为节点，那到底怎么标记不同的使用者呢？也是通过用户名注册吗？实则不然。在区块链系统中，这个地方的设计很有意思，是通过一种密码算法来实现的，具体来说是通过一种叫公开密钥算法的机制来实现的。我们知道，对于一种密码算法来说，无论算法过程是什么样的，都会有一个密钥。而公开密钥算法拥有一对（也就是两个）密钥，跟虎符一样，是彼此配合使用的，可以互相用来加解

密。其中一个叫私钥，另外一个叫公钥，公钥可以公开给别人，私钥要自己保管好。在区块链系统中，公钥就是用来用户身份识别的，一般不会直接使用公钥，因为不容易让人记住。公钥往往都比较长，实际处理的时候都会进行转换，比如取得公钥的最后20个字节或者经过一系列更复杂的转换，最后得到一个称为“地址”的转换结果，这个“地址”就能代表一个用户。

为什么在区块链系统中要用这么一个奇怪的用户身份表示方法呢？似乎看起来除了有些创意外，也没特别的用处。这里我们就得再介绍下这个公开密钥算法的特别能力。之前提到说这种算法有两个密钥，那么这两个密钥是怎么配合工作的呢？我们来简单说明一下：用公钥加密的数据必须用对应的私钥来解密，而用私钥加密（通常称为“签名”）的数据必须用对应的公钥来解密。这个特点可是能发挥很大用处的，就如上述的例子中，如果张三要发送给李四一张支票，那怎么传送呢？就这么发过去，会被那个记账的人拿到，风险可就大了。于是张三想了一个办法，他在支票上用李四的公钥加了个密，然后再签上自己的名字（使用自己的私钥签名），这个时候其他人就算拿到支票也没用，因为只有李四才有自己的私钥，也只有李四才能解开这张支票来使用。这种功能设计在区块链系统中称为“脚本系统”。

现在我们知道了，区块链的技术理念，其实就是大家共同来参与记账，通过一种规则不断地选出账务打包者，其他节点接收验证，并且每个用户都有一对密钥表示自己，通过脚本系统的功能实现在公共网络中定向发送有价值的数据。

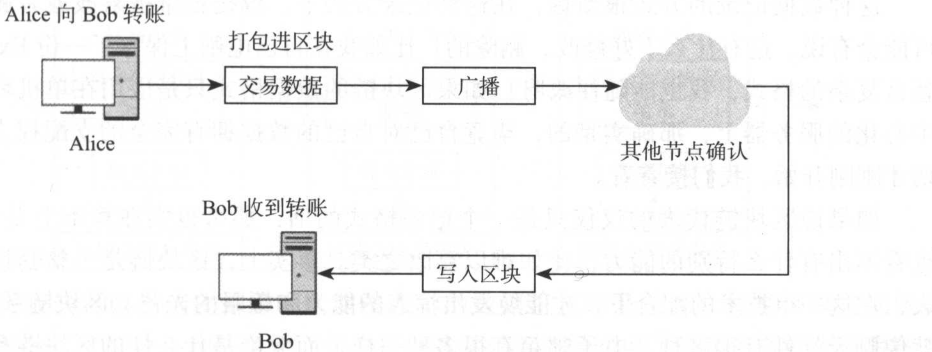
1.1.3 一般工作流程

通过上面的例子，相信读者朋友对区块链已经有了基本认识。区块链系统有很多种，第一个应用区块链技术的软件就是比特币，事实上区块链的概念就是比特币带出来的。到现在为止，已经出现了相当多的基于区块链技术的衍生系统，比如闪电网络、公证通、以太坊、超级账本项目等。每一类系统都有自己的特点，例如汽车设计，有的设计成跑车，有的设计成运输车，有的设计成商务车，但是有一点，无论是什么类型的车，它的工作方式或者说工作流程都是类似的，在本质上它们都是同一类技术结构的产物。在这一小节，我们从一般性的角度阐述一下区块链系统的工作流程，为了便于说明，我们会选取一些场景例子。

我们先来看一个转账交易的流程。转账交易本质上就是发送一笔数据，这个数据可以表示为资产，也可以表示为订单或者其他各种形式的数据，我们看一下下面的图示。

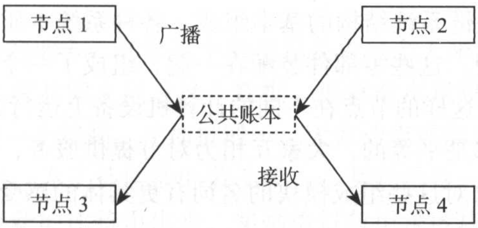
从图中我们可以看到，整个数据的发送过程其实还是很简单的，数据发送出去后，会被打包进区块，然后广播出去给所有的节点确认，确认没有问题后就写入到各自的本地区块链账本中，当网络中的大多数节点都确认写入后，这个转账过程就算是完成了。有朋友可能会问，在这种分布式的网络中，怎么能知道是被大多数节点确认写入了呢？这里并没有什么服务器登记呀？这个问题我们先留着，在下面讲到区块链分类的时候会有详细的解

释，大家可以先思考一下。



这个工作流程图是有代表性的，其他各种系统都是在这个基础上进行衍生和扩展。比如有些会增加身份认证功能，以确保只有符合身份验证的用户才能发送数据；有些则扩展交易数据的表达能力，不但能用来表示一般的交易转账，还能表示更复杂的商业逻辑。各种应用很多，但是万变不离其宗。

实际上，说一千道一万，整个区块链网络，就是大家共同来维护一份公共账本。注意了，这个公共账本是一个逻辑上的概念，每个节点各自都是独立维护自己账本数据的，而所谓的公共账本，是说各自的账本要保持一致，保持一致的部分就是公共账本，我们看下图示：



如图所示，有些节点在广播新的数据，有些节点在接收数据，大家共同维护一个账本，确保达成一致。区块链技术其实就是围绕如何保持数据的一致、如何让这个公共账本的数据不被篡改来展开的。为了解决这些问题，区块链技术拥有一套技术栈，我们通过以下章节来阐述。

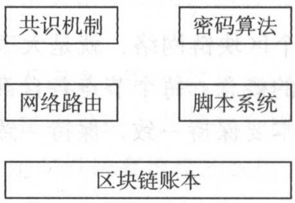
1.2　区块链技术栈

区块链本身只是一种数据的记录格式，就像我们平常使用的 Excel 表格、Word 文档一样，按照一定的格式将我们的数据存储在电脑上。与传统的记录格式不同的是，区块链是将产生的数据按照一定的时间间隔，分成一个个的数据块记录，然后再根据数据块的先后关系串联起来，也就是所谓的区块链了。按照这种规则，沿着时间线不断增加新的区块，

就好像是时光记录仪一样，记录下发生的每一笔操作。

这种数据记录的方式很新颖，在这种记录方式下，数据很难被篡改或者删除。有朋友可能会有说，这有什么不好修改、删除的！比如我在自己电脑上保存了一份 Excel 数据，再怎么复杂的格式，我也能随便改呀！如果区块链的数据格式只是应用在单机环境或者一个中心化的服务器上，那确实是的，毕竟自己对自己的数据拥有完全的支配权力。然而，一切才刚刚开始，我们接着看。

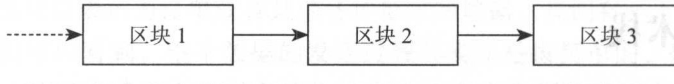
如果说区块链代表的仅仅只是一个记录格式的话，那么也实在算不上是伟大的发明，也看不出有什么特别的能力，比如难以篡改之类。事实上，区块链是一整套技术组合的代表，在这一组技术的配合下，才能焕发出惊人的能力和耀眼的光芒。区块链系统有很多种，就像聊天软件有很多种，电子邮箱有很多种一样，而无论是什么样的区块链系统，其技术部件的组合都是类似的。就像汽车基本都是由发动机、底盘、车身、电器四大部件组成的，计算机都是由 CPU、存储器、输入 / 输出设备组成的，不管是比特币、莱特币、以太坊还是其他，核心结构和工作原理都是共同的。我们就来看看最基本的技术组合都有哪些：



如图所示，这是区块链系统结构的基本组成，各种系统本质上都是在这个经典结构之上直接实现或者扩展实现。这些零部件装配在一起，组成了一个区块链软件，运行起来后就称之为一个节点，多个这样的节点在不同的计算机设备上运行起来，就组成了一个网络。在这个网络中每个节点都是平等的，大家互相为对方提供服务，这种网络被称为点对点的“对等网络”。为了让大家对这些组成模块的名词有更具体的感受和理解，我们一一来解释一下。

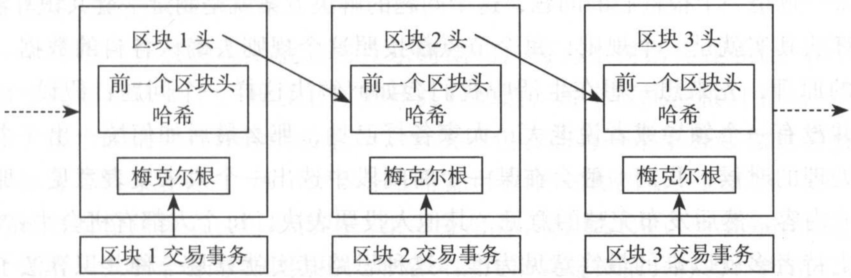
1. 区块链账本

如上所述，它表示一种特有的数据记录格式。区块链，就是“区块 + 链”，所谓的区块就是指数据块的意思，每一个数据块之间通过某个标志连接起来，从而形成一条链，我们看下示意图：



如图所示，一个区块一个区块地衔接。大家可以发现在生活中有很多相似的记录方式，比如企业的会计账簿，每个月会计将记账凭证汇总为账簿并且月结，这样一段时间下来，就按月形成了一个连续账簿，每个月的数据就相当于区块，区块与区块之间通过年月串联起来。以比特币来说，大约是每 10 分钟产生一个区块，区块中主要包含了交易事务数据以

及区块的摘要信息。我们看下比特币中区块链数据的组成示意图：



通过上图我们可以看到比特币中区块链账本的数据组成以及关系，并且可以看到区块数据在逻辑上分成了区块头和区块体，每个区块头中通过梅克尔根[⊖]关联了区块中众多的交易事务，而每个区块之间通过区块头哈希值（区块头哈希值就是一个区块的身份证号）串联起来。这是一个很有趣的数据格式，它将连续不断发生的数据分成了一个一个的数据块。在下载同步这些数据的时候，可以并行地从各个节点来获得，无论数据先后，到达本地后再根据身份证号组装起来就行。另外，这是一种链条格式，链条最大的特点就是一环扣一环，很难从中间去破坏。比如，有人篡改了中间的 2 号区块，那么就得同时把 2 号区块后续的所有区块都更改掉，这个难度就大了。在区块链系统中，一个节点产生的数据或者更改的数据要发送到网络中的其他节点接受验证，而其他节点是不会验证通过一个被篡改的数据的，因为跟自己的本地区块链账本数据匹配不起来，这也是区块链数据不可篡改的一个很重要的技术设计。

这种格式还有个巧妙的地方，如果这个数据总是由一个人来记录的，那自然也没什么，但是如果放到网络中，大家共同来记录这个数据，那就有点意思了，每个区块数据由谁来记录或者说打包，可以有一个规则。比如掷骰子，大家约定谁能连续 3 次掷出 6，那就让他来记录下一个区块的数据，为了补偿他的劳动投入，奖励给他一些收益。比特币正是使用了这样的原理来不断发行新的比特币出来，奖励给打包记录区块数据的那个人的比特币就是新发行的比特币。

2. 共识机制

所谓共识，就是指大家都达成一致的意思。在生活中也有很多需要达成共识的场景，比如开会讨论，双方或多方签订一份合作协议等。在区块链系统中，每个节点必须要做的事情就是让自己的账本跟其他节点的账本保持一致。如果是在传统的软件结构中，这几乎就不是问题，因为有一个中心服务器存在，也就是所谓的主库，其他的从库向主库看齐就行了。在实际生活中，很多事情人们也都是按照这种思路来的，比如企业老板发布了一个通知，员工照着做。但是区块链是一个分布式的对等网络结构，在这个结构中没有哪个节

⊖ 梅克尔根也称为“梅克尔根哈希值”，具体概念后续有详细介绍，暂且可以认为就是一个区块中所有交易事务的集体身份证号。

点是“老大”，一切都要商量着来。在区块链系统中，如何让每个节点通过一个规则将各自的数据保持一致是一个很核心的问题，这个问题的解决方案就是制定一套共识算法。

共识算法其实就是一个规则，每个节点都按照这个规则去确认各自的数据。我们暂且抛开算法的原理，先来想一想在生活中我们会如何解决这样一个问题：假设一群人开会，这群人中并没有一个领导或者说老大，大家各抒己见，那么最后如何统一出一个决定出来呢？实际处理的时候，我们一般会在某一个时间段中选出一个人来发表意见，那个人负责汇总大家的内容，然后发布完整的意见，其他人投票表决，每个人都有机会来做汇总发表，最后谁的支持者多就以谁的最终意见为准。这种思路其实就算是一种共识算法了。然而在实际过程中，如果人数不多并且数量是确定的，那还好处理些，如果人数很多而且数量也不固定，那我们就很难让每个人都去发表意见然后再来投票决定了，这样效率就太低了。我们需要通过一种机制筛选出最有代表性的人，在共识算法中就是筛选出具有代表性的节点。

如何筛选呢？其实就是设置一组条件，就像我们筛选运动员，筛选尖子生一样，给一组指标让大家来完成，谁能更好地完成指标，谁就能有机会被选上。在区块链系统中，存在着多种这样的筛选方案，比如 PoW（Proof of Work，工作量证明）、PoS（Proof of Stake，权益证明）、DPoS（Delegate Proof of Stake，委托权益证明）、PBFT（Practical Byzantine Fault Tolerance，实用拜占庭容错算法）等，各种不同的算法，其实就是不同的游戏玩法，限于篇幅，这里暂不进行算法过程的详述，大家只要知道这些都是一种筛选算法就行了。区块链系统就是通过这种筛选算法或者说共识算法来使得网络中各个节点的账本数据达成一致的。

3. 密码算法

密码算法的应用在区块链系统中是很巧妙的，应用的点也很多，我们在这里不详细介绍密码算法的原理，就从几个很关键的应用来介绍一下。

首先我们回顾下区块链账本格式。通过上述讲解我们已经知道，区块链账本就是连接起来的一个个区块。那么到底是通过什么来连接的呢？学过数据结构的朋友都知道，在数据结构中，有一种变量叫指针，它是可以用来指向某个数据的地址的。那么区块的连接是不是通过这样的数据地址呢？生活中的地址连接例子很多，比如路牌、门牌等。然而，区块之间的连接，往往都不是靠数据地址来关联的，而是靠一种叫作哈希值的数据来关联的。什么叫哈希值？这是通过密码算法中的哈希算法计算得出的。哈希算法可以通过对一段数据计算后得出一段摘要字符串，这种摘要字符串与原始数据是唯一对应的。什么意思呢？如果对原始数据进行修改，哪怕只是一点点修改，那么计算出来的哈希值都会发生完全的变化。区块链账本对每个区块都会计算出一个哈希值，称为区块哈希，通过区块哈希来串联区块。这有一个很好的作用就是，如果有人篡改了中间的某一个区块数据，那么后面的区块就都要进行修改，这个时候并不是简单地修改一下后面区块的地址指向就能结束的，

由于后面的区块是通过区块哈希来指向的，只要前面的区块发生变动，这个区块哈希就无效了，就指不到正确的区块了。

另外一个对密码算法的应用就是梅克尔树结构，梅克尔树结构在 3.1.3 节中有详细介绍，我们这里先初步认识下。通过上述解释我们知道，每个区块会被计算出一个哈希值。实际上，除了整个区块会被计算哈希值外，区块中包含的每一笔事务数据也会被计算出一个哈希值，称为“事务哈希”，每一个事务哈希都可以唯一地表示一个事务。对一个区块中所有的事务进行哈希计算后，可以得出一组事务哈希，再通过对这些事务哈希进行加工处理，最终会得出一棵哈希树的数据结构。哈希树的顶部就是树根，称为“梅克尔根”。通过这个梅克尔根就可以将整个区块中的事务约束起来，只要区块中的事务有任何改变，梅克尔根就会发生变化，利用这一点，可以确保区块数据的完整性。

当然，密码算法在区块链系统中的应用还远不止这些，比如通过密码算法来创建账户地址、签名交易事务等，这些应用在后续章节中会逐步介绍。

4. 脚本系统

脚本系统在区块链中是一个相对抽象的概念，也是极其重要的一个功能，可以说区块链系统之所以能形成一个有价值的网络，依靠的就是脚本系统，它就像是发动机一样，驱动着区块链系统不断进行着各种数据的收发。所谓脚本，就是指一组程序规则。在区块链系统中，有些系统中的程序规则是固定的，比如在比特币系统中，只能进行比特币的发送与接收，这个发送与接收的过程就是通过实现在比特币中的一组脚本程序来完成的；而有些系统是允许用户自行编写一组程序规则的，编写好后可以部署到区块链账本中，这样就可以扩展区块链系统的功能，比如以太坊就是通过实现一套可以自定义功能的脚本系统，进而实现了智能合约的功能。

脚本系统使得在区块链中可以实现各种各样的业务功能。本来大家只是通过区块链来记财务账的，通过脚本系统，大家可以使用区块链来记录各种各样的数据，比如订单、众筹账户、物流信息、供应链信息等，这些数据一旦可以记录到区块链上，那么区块链的优点就能够被充分地发挥出来。有关脚本系统的具体使用和开发，大家可以通过后续的第 6 ～ 8 章来理解。

5. 网络路由

这个功能模块比较简单。区块链系统是一个分布式的网络，这些网络中的节点如何来彼此进行连接通信呢？依靠的就是网络路由功能。前面我们说到，张三、李四、王五、赵六是通过彼此介绍来认识的，这个其实就是网络路由的雏形了。在分布式的网络结构中，不存在一个指定的服务器，大家没法通过一个服务器来直接交换彼此的身份信息，就只能依靠彼此联系并传播信息。在区块链系统中，这个功能一般会定义成一种协议，称为“节点发现协议”。

除了要发现节点外，更重要的一个功能就是同步数据。节点要保持自己的账本数据是

最新的，就必须要时时更新自己的数据。从哪更新呢？既然没有服务器来下载，那就是通过邻近的节点了。通过向邻近节点发送数据请求来获得最新的数据，节点彼此都充当服务者和被服务者，通过这种方式，网络中的每一个节点都会在某一个时刻达成数据上的一致。

网络路由可以说是区块链系统中的触角，通过大量的触角将每个节点连入网络，从而形成一个功能强大的区块链共识网络。

1.3 区块链分类与架构

通过上述了解，我们知道区块链系统实际上就是一个维护公共数据账本的系统，一切技术单元的设计都是为了更好地维护好这个公共数据账本。通过共识算法达成节点的账本数据一致；通过密码算法确保账本数据的不可篡改性以及数据发送的安全性；通过脚本系统扩展账本数据的表达范畴。我们甚至可以认为，区块链系统实际上就是一种特别设计的数据库系统或者说分布式数据库系统，在这个数据库中可以存储数字货币，也可以存储逻辑更复杂的智能合约，以及范围更加广阔的各种业务数据。在区块链系统的发展过程中，也经历了这样一个阶段，从比特币开始，早期的区块链系统都是面向数字货币的，如比特币、莱特币等，这个阶段我们可以认为区块链系统是一个支持数字货币合约的系统；之后便出现了更加灵活的，能够支持自定义智能合约的系统，其代表作是以太坊，可以认为以太坊就是对比特币这样的数字货币系统的扩展，不过以太坊仍然内置了对数字货币的支持，延续了比特币系统的金融特征，也使得以太坊的应用更多面向金融范畴；再之后的代表就是超级账本项目，尤其是其中的 Fabric 子项目，在这个系统中，超越了对金融范畴的应用，支持各个领域的数据定义，我们分别将这三个阶段称为区块链系统的 1.0、2.0、3.0 架构时期。为了让大家对发展过程中的区块链系统有一个整体的概念，在本节中，我们来描述一下通常的区块链系统的架构，并站在不同的角度对区块链系统进行分类。

1.3.1 区块链架构

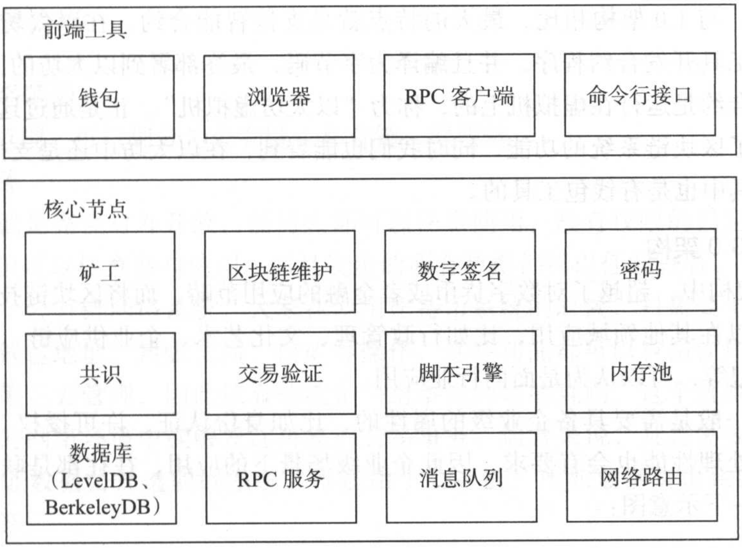
1. 区块链 1.0 架构

如上所述，这个阶段区块链系统主要是用来实现数字货币的，我们看一下示意图。

如图所示，在整个架构中，分为核心节点和前端工具，这里提一下核心节点中“矿工”功能。矿工在 1.0 架构的系统中，主要是承担两个任务：

第一个是通过竞争获得区块数据的打包权后将内存池（发送在网络中但是还没有确认进区块的交易数据，属于待确认交易数据）中的交易数据打包进区块，并且广播给其他节点；

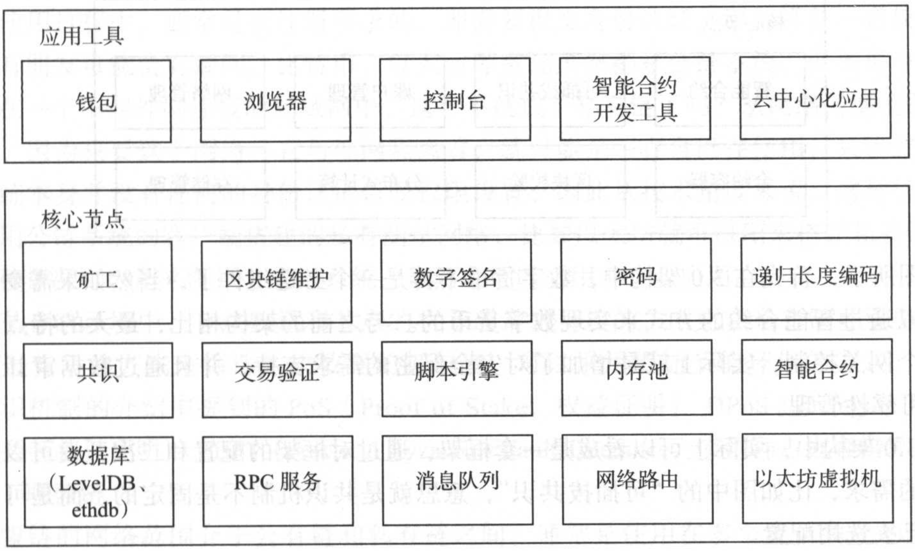
第二个是接受系统对打包行为的数字货币奖励，从而系统通过这种奖励方式完成新增货币的发行。



在前端工具中，最明显的就是钱包工具，钱包工具是提供给用户管理自己账户地址以及余额的；浏览器则用来查看当前区块链网络中发生的数据情况，比如最新的区块高度、内存池的交易数、单位时间的网络处理能力等；RPC 客户端和命令行接口都是用来访问核心节点的功能的，在这个时候，核心节点就相当于一个服务器，通过 RPC 服务提供功能调用接口。

2. 区块链 2.0 架构

区块链 2.0 架构的代表产品是以太坊，因此我们可以套用以太坊的架构来说明，先看下示意图：

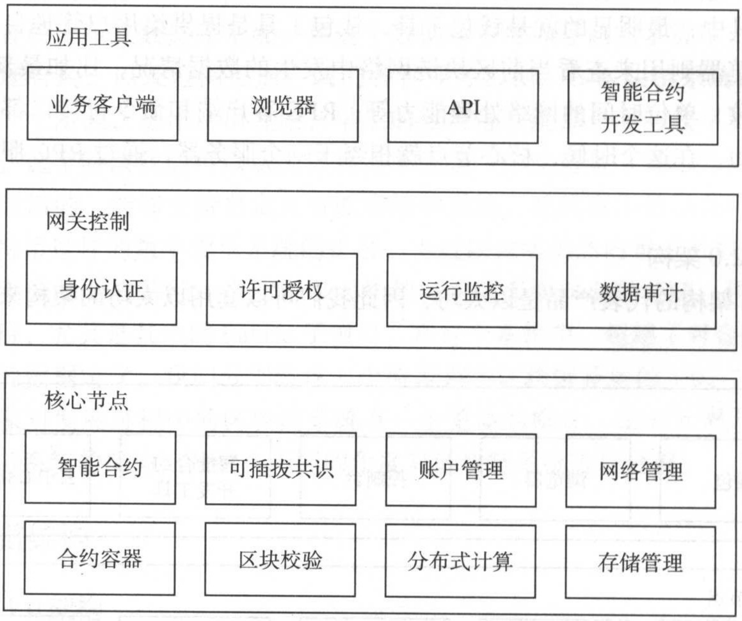


如图所示，与1.0架构相比，最大的特点就是支持智能合约，在以太坊中，我们使用智能合约开发工具开发合约程序，并且编译为字节码，最终部署到以太坊的区块链账本中。部署后的智能合约是运行在虚拟机上的，称为“以太坊虚拟机”。正是通过这样的智能合约的实现，扩展了区块链系统的功能，同时我们也能看到，在以太坊中还是支持数字货币的，因此在应用工具中也是有钱包工具的。

3. 区块链3.0架构

在3.0的架构中，超越了对数字货币或者金融的应用范畴，而将区块链技术作为一种泛解决方案，可以在其他领域应用，比如行政管理、文化艺术、企业供应链、医疗健康、物联网、产权登记等，可以认为是面向行业应用。

行业应用一般是需要具备企业级的属性的，比如身份认证、许可授权、加密传输等，并且对数据的处理性能也会有要求，因此企业级场景下的应用，往往都是联盟链或者私有链。我们来看一下示意图：



如图所示，首先在3.0架构中，数字货币不再是一个必选组件了，当然如果需要，我们也是可以通过智能合约的方式来实现数字货币的。与之前的架构相比，最大的特点就是增加了一个网关控制，实际上就是增加了对安全保密的需求支持，并且通过数据审计加强对数据的可靠性管理。

在3.0架构中，实际上可以看成是一套框架，通过对框架的配置和二次开发可以适应各行各业的需求，比如图中的“可插拔共识”，意思就是共识机制不是固定的，而是可以通过用户自己去选用配置。

1.3.2　区块链分类

1. 根据网络范围

根据网络范围，可以划分为公有链、私有链、联盟链。

（1）公有链

所谓公有就是完全对外开放，任何人都可以任意使用，没有权限的设定，也没有身份认证之类，不但可以任意参与使用，而且发生的所有数据都可以任意查看，完全公开透明。比特币就是一个公有链网络系统，大家在使用比特币系统的时候，只需要下载相应的软件客户端，创建钱包地址、转账交易、挖矿等操作，这些功能都可以自由使用。公有链系统由于完全没有第三方管理，因此依靠的就是一组事先约定的规则，这个规则要确保每个参与者在不信任的网络环境中能够发起可靠的交易事务。通常来说，凡是需要公众参与，需要最大限度保证数据公开透明的系统，都适用于公有链，比如数字货币系统、众筹系统、金融交易系统等。

这里要注意，在公有链的环境中，节点数量是不固定的，节点的在线与否也是无法控制的，甚至节点是不是一个恶意节点也不能保证。我们在讲解区块链的一般工作流程的时候，提到过一个问题，在这种情况下，如何知道数据是被大多数的节点写入确认的呢？实际在公有链环境下，这个问题没有很好的解决方案，目前最合适的做法就是通过不断地去互相同步，最终网络中大多数节点都同步一致的区块数据所形成的链就是被承认的主链，这也被称为最终一致性。

（2）私有链

私有链是与公有链相对的一个概念，所谓私有就是指不对外开放，仅仅在组织内部使用的系统，比如企业的票据管理、账务审计、供应链管理等，或者一些政务管理系统。私有链在使用过程中，通常是有注册要求的，即需要提交身份认证，而且具备一套权限管理体系。有朋友可能会有疑问，比特币、以太坊等系统虽然都是公链系统，但如果将这些系统搭建在一个不与外网连接的局域网中，这个不就成了私有链了吗？从网络传播范围来看，可以算，因为只要这个网络一直与外网隔离着，就只能是一直自己在使用，只不过由于使用的系统本身并没有任何的身份认证以及权限设置，因此从技术角度来说，这种情况只能算是使用公链系统的客户端搭建的私有测试网络，比如以太坊就可以用来搭建私有链环境，通常这种情况可以用来测试公有链系统，当然也可以适用于企业应用。

在私有链环境中，节点数量和节点的状态通常是可控的，因此在私有链环境中一般不需要通过竞争的方式来筛选区块数据的打包者，可以采用更加节能环保的方式，比如在上述共识机制的介绍中提到的 PoS（Proof of Stake，权益证明）、DPoS（Delegate Proof of Stake，委托权益证明）、PBFT（Practical Byzantine Fault Tolerance，实用拜占庭容错算法）等。

（3）联盟链

联盟链的网络范围介于公有链和私有链之间，通常是使用在多个成员角色的环境中，

比如银行之间的支付结算、企业之间的物流等，这些场景下往往都是由不同权限的成员参与的，与私有链一样，联盟链系统一般也是具有身份认证和权限设置的，而且节点的数量往往也是确定的，对于企业或者机构之间的事务处理很合适。联盟链并不一定要完全管控，比如政务系统，有些数据可以对外公开的，就可以部分开放出来。

由于联盟链一般用在明确的机构之间，因此与私有链一样，节点的数量和状态也是可控的，并且通常也是采用更加节能环保的共识机制。

2. 根据部署环境

（1）主链

所谓主链，也就是部署在生产环境的真正的区块链系统，软件在正式发布前会经过很多内部的测试版本，用于发现一些可能存在的Bug，并且用来内部演示以便于查看效果，直到最后才会发布正式版。主链，也可以说是由正式版客户端组成的区块链网络，只有主链才是会被真正推广使用的，各项功能的设计也都是相对最完善的。另外，有些时候，区块链系统会由于种种原因导致分叉，比如挖矿的时候临时产生的小分叉等，此时将最长的那条原始的链条称为主链。

（2）测试链

这个很好理解，就是开发者为了方便大家学习使用而提供的测试用途的区块链网络，比如比特币测试链、以太坊测试链等。当然，倒也不是说非得是区块链开发者才能提供测试链，用户也可以自行搭建测试网络。测试链中的功能设计与生产环境中的主链是可以有一些差别的，比如主链中使用工作量证明算法进行挖矿，在测试链中可以更换算法以便更方便地进行测试使用。

3. 根据对接类型

（1）单链

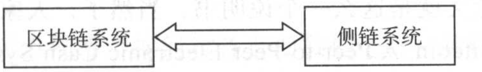
能够单独运行的区块链系统都可以称为“单链”，例如比特币主链、测试链；以太坊主链、测试链；莱特币的主链、测试链；超级账本项目中的Fabric搭建的联盟链等，这些区块链系统拥有完备的组件模块，自成一个体系。大家要注意了，对于有些软件系统，比如基于以太坊的众筹系统或者金融担保系统等，这些只能算是智能合约应用，不能算是一个独立的区块链系统，应用程序的运行需要独立的区块链系统的支撑。

（2）侧链

侧链属于一种区块链系统的跨链技术，这个概念主要是由比特币侧链发起的。随着技术发展，除了比特币，出现了越来越多的区块链系统，每一种系统都有自己的优势特点，如何将不同的链结合起来，打通信息孤岛，彼此互补呢？侧链就是其中的一项技术。

以比特币来说，比特币系统主要是设计用来实现数字加密货币的，且业务逻辑也都固化了，因此并不适用于实现其他的功能，例如金融智能合约、小额快速支付等。然而比特币是目前使用规模最大的一个公有区块链系统，在可靠性、去中心化保证等方面具有相当

的优势，那么如何利用比特币网络的优势来运行其他的区块链系统呢？可以考虑在现有的比特币区块链之上，建立一个新的区块链系统，新的系统可以具备很多比特币没有的功能，比如私密交易、快速支付、智能合约、签名覆盖金额等，并且能够与比特币的主区块链进行互通，简单来说，侧链是以锚定比特币为基础的新型区块链。锚定比特币的侧链，目前有 ConsenSys 的 BTCRelay、Rootstock 和 BlockStream 的元素链等。大家要注意，侧链本身就是一个区块链系统，并且侧链并不是一定要以比特币为参照链，这是一个通用的技术概念，比如以太坊可以作为其他链的参照链，也可以本身作为侧链与其他的链去锚定。实际上，抛开链、网络这些概念，就是不同的软件之间互相提供接口，增强软件之间的功能互补，我们看下侧链的示意图：



通过这个简单的示意图，我们可以看到，区块链系统与侧链系统本身都是一个独立的链系统，两者之间可以按照一定的协议进行数据互动，通过这种方式，侧链能起到一个对主链功能扩展的作用，很多在主链中不方便实现的功能可以实现在侧链中，而侧链再通过与主链的数据交互增强自己的可靠性。

（3）互联链

如今我们的生活可以说几乎已经离不开互联网了，仅仅互通互联，带来的能量已经如此巨大。

区块链也是这样，目前各种区块链系统不断涌现，有的只是实现了数字货币，有的实现了智能合约，有的实现了金融交易平台，有些是公有链，有些是联盟链，等等。这么多的链，五彩缤纷，功能各异，脑洞大开，不断刷新着更新颖的应用玩法。那么，这些链系统如果能够彼此之间互联会发生些什么样的化学反应呢？与传统软件不同的是，区块链应用拥有独特的性质，比如数据不可篡改性、完整性证明、自动网络共识、智能合约等，从最初的数字货币到未来可能的区块链可编程社会，这些不单单会改变生活服务方式，还会促进社会治理结构的变革，**如果说每一条链都是一条神经的话，一旦互联起来，就像是神经系统一般，将会给我们的社会发展带来更新层次的智能化**。

另外，从技术角度来讲，区块链系统之间的互联，可以彼此互补，每一类系统都会有长处和不足之处，彼此进行功能上的互补，甚至可以彼此进行互相的验证，可以大大加强系统的可靠性以及性能。

1.4　一切源自比特币

当我们坐在飞机上，开启一段美妙的旅程时，可否会想起当初的莱特兄弟；当我们坐在高铁上，享受着高效的城市穿梭时，可否会想起当初的蒸汽机；当我们住在舒适的房屋里，享受着安心的睡眠时，可否会想起当初的茅草房。是的，这个世界给了我们很多原材

料，我们使用原材料，制造出了一个又一个工具，并以此改造这个世界，改善我们的生活。区块链，便是这样的一个改造世界的原材料，而有人用它制造出了第一个工具，它的名字叫比特币。

1.4.1 比特币技术论文介绍

通常，在介绍一个比较重量级的人物的时候，我们常常会在他的名字前面加上很多定语，比如某著名歌唱家、慈善大使、两届 ×× 奖获得者等，然后最后才报出名字，为的就是让大家竖起耳朵听明白，这个牛人都能干些什么。而在介绍一个物件的时候，比如一辆汽车，我们就不会这么说了，因为能把人说睡着了，一个东西嘛，写个说明不就完了，一目了然。

那么，比特币技术论文就是这么一个说明书，当然了，人家这份说明书可是有正式名字的，人家的大名叫《Bitcoin: A Peer-to-Peer Electronic Cash System》，翻译过来叫《比特币：一种点对点的电子现金系统》。这篇技术论文通常称为比特币白皮书，因为它基本就是宣告了比特币的诞生。严格地说，是理论上宣告了比特币的诞生。这份文件是在 2008 年 11 月由一个叫 Satoshi Nakamoto（中本聪）的人发布的。当然了，并不是发布在什么知名论坛或者学术期刊上，而是发布在一个小众的密码学讨论小组。在这份白皮书发布后的第二年，也就是 2009 年 1 月 3 日，比特币软件就正式启动运行了，也就是在这个时候，世界上第一个区块链数据诞生了，而这个由中本聪构造出来的第一个区块，也称为创世区块或者上帝区块，代表神话中创世元灵的意思。从此以后，比特币以及由比特币技术衍生出来的各种应用就一发不可收拾，开启了互联网应用的一个新纪元。

回到这个白皮书上来，注意看它的标题，有两个关键字："点对点"和"电子现金"。有朋友说了，这俩词压根就没提什么区块链！别着急，看人看眼睛，读文读标题，咱们先来解释一下。"点对点"，就是指这个软件不需要一个特定的服务器，比如我们登录 QQ 就需要连接腾讯的 QQ 服务器，登录支付宝就需要连接阿里巴巴的支付宝服务器，倘若这些服务器关闭或者出个问题什么的，那就没法正常使用这些软件了。2015 年 5 月，杭州电信光缆被施工队不慎挖断，直接导致通过这些光缆联网的支付宝服务器断网，影响了正常的运行。而点对点的网络结构，并不依赖于某一个或者某一群特定的服务器，相当于人人都是服务器，人人也都是使用者。再来看"电子现金"，顾名思义，现金嘛，就是钱或者货币的意思，也就是说这份白皮书，介绍的是一种数字货币系统，这个系统的运行不依赖于某些特定服务器，而是通过点对点网络（P2P）结构来运行的。相信有些读者朋友看到这里还是会有些懵，不要紧，毕竟咱们才刚看到标题嘛，有个概念就行了。

翻开白皮书正文可以发现，整个篇幅主要介绍了几个关键点。

（1）简介

提出了一个场景设想：如何不通过一个所谓的权威第三方结构（比如银行），来构建一个可信的交易网络呢？中本聪的语文还是不错的，先抛出个问题给你玩玩，然后吸引你继续看下去。

（2）交易

描述了一种通过密钥签名进行交易验证的方式，实际上就是计算机密码学在比特币中的应用。我们在银行转账交易用什么来证明自己呢？是通过账户和密码，必要的时候还可以通过身份证确认。而在比特币系统中没有银行这样一个角色，那靠什么来确定身份呢？只有靠现代计算机密码学技术。当然，密码学技术在比特币中的应用并不只是用来证明身份，是贯穿在各个环节的，可以说，密码学技术就是比特币系统的骨骼。

（3）时间戳服务器

这部分提到了区块以及通过时间戳运算连接成一条链的概念，这也是区块链概念的来源，同时在这里也说明了比特币数据的存储方式。

（4）工作量证明

这部分介绍了一种点对点网络中如何对各自的数据进行一致性确认的算法。为什么叫工作量证明呢？因为这种算法很消耗 CPU 的算力，等于人们干活一样，是要付出工作劳动的。

（5）网络

比特币软件是一种网络软件，而且是一个不依靠某个服务器来交换数据的网络软件。那么一个个节点之间，如何确认一笔笔交易数据呢？这部分介绍了交易确认的过程，这个实际上就是比特币网络的应用协议，跟日常使用的邮件收发协议、文件传输协议、超文本传输协议等，是一个层面上的。

（6）激励

激励就是奖励的意思，你干了活，得到一笔奖金，哇，好开心！就会继续努力干活，这就是激励。比特币软件的数据一致性确认是需要耗费 CPU 算力的，那凭什么有人愿意来耗费这些个算力，白干活吗？当然不是，系统会奖励给你比特币，还有别人交易的手续费。（有人会问，那我为什么没被奖励过啊？别急，在 1.4.3 节中会有详述。）

（7）回收硬盘空间

比特币系统从创世区块开始，大约每 10 分钟产生一个区块，也意味着区块链账本的“体积”会一直增长。事实上写作本书的时候，已经超过了 120GB，只要比特币网络一直存在，数据就会一直增长。实际上，只有运行全功能节点的客户端才会一直保持完整的区块链数据，这些在 1.4.2 中会有详述。这里提出了一个思路，删除过老的一些交易数据，同时不破坏区块的随机哈希值，通过这种方法压缩区块数据。

（8）简化的支付确认

上述提到了，比特币客户端的数据量很大，这么一来，等于不管是用比特币系统干什么都要带上大量的数据，这岂不是很不方便，而且也会限制在其他一些终端（比如手机）上的使用。这部分提出了一个模型，这个模型主要是为比特币的支付服务的。在这个模型下实现的比特币支付功能并不需要携带那么庞大的数据，而只需要保留体积相对很小的区块头，具体细节可以查看 3.1.3 节。

（9）价值的组合与分割

这部分介绍的是比特币中的交易事务组成方式。①什么叫价值？在比特币系统中，价值就是比特币。②什么叫组合？比如我口袋里有5枚1元硬币，1张2元纸币，1张10元纸币，我要给你5块钱，怎么给呢？我可以给你5枚1元硬币，也可以给你3枚1元硬币加上1张2元纸币，这就是不同的组合。③什么叫分割？分割其实就是转出的意思，我通过不同的组合，构成了总计5元的金额，然后转出给你，这个过程就是价值的组合和重新分割。在这个例子中，还有一张10元的，假如我直接转了你10元，那会怎样？这就需要找零5元了，找零其实也是一种重新价值分割。

（10）隐私

作为一个货币系统，保密性也就是隐私毫无疑问是人人都会关心的。传统的体系，完全是依赖比如银行这个第三方的保护，大家相信银行，银行也设立了各种管理制度和方法来防止账户和交易信息的泄密。比特币系统则不同，它不依赖谁，每个人在比特币系统中也不用登记什么身份证、名称、性别等，就是一个地址，谁也不知道地址后面代表的是谁，而且，只要你需要，可以自己创建任意多个地址（你到银行去开任意多个户试试！），这使得比特币系统中的交易带有很大的匿名性和隐秘性。

（11）计算

这部分主要是站在概率统计的角度计算了一下攻击者成功的概率，以及经过多少个区块后还能攻击成功的概率，计算过程这里不赘述。

白皮书的内容就介绍到这里了。刚刚接触比特币、区块链这些概念的朋友，或许还是一头雾水吧！没关系，我们在下面的章节会有详细的解释。毕竟，能够只通过一份白皮书就完全明白比特币设计的人，或许只有中本聪这个“大神”了。

小提示　白皮书的原文可以在 https://bitcoin.org/bitcoin.pdf 查看，感兴趣的朋友可以阅读一下，英文不那么擅长的朋友，可以到巴比特网站（著名的区块链资讯与技术服务网站）上查看中文版，地址是 http://www.8btc.com/wiki/bitcoin-a-peer-to-peer-electronic-cash-system。读明白了这份说明书，基本也就理解了比特币的原理，也就入了区块链这个“坑”（或者说这个“门”）了。

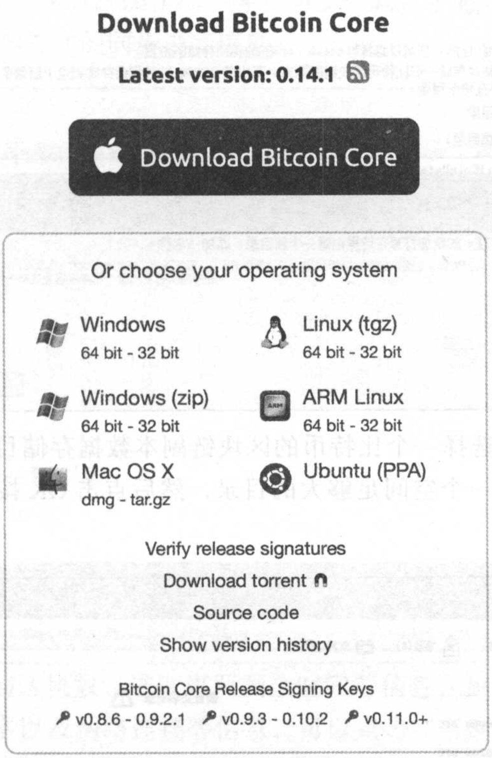
1.4.2 比特币核心程序：中本聪客户端

1. 客户端程序介绍

我们知道，比特币其实就是一个软件，既然是软件，那还是百闻不如一见，看看到底长什么样。大家可以到 https://bitcoin.org/en/download 这个地址去下载客户端程序，可以看到，网站提供了多种操作系统的运行版本，选择自己需要的版本下载安装即可运行了，就能看到庐山真面目啦。

在具体介绍之前，咱们先说明一下，为什么这个程序叫比特币核心程序，难道还有非核心程序？我们在上述提供的下载页面上，可以看到比特币程序的名字叫 Bitcoin core，这个翻译过来就是比特币核心的意思，这是最经典，也是中本聪一开始发布的那一支程序版本，这个版本也是使用人数最多的。可问题是，比特币程序是开源的，任何一个人或者组织都可以根据需要去修改源码发布出一个新的版本，事实上经过多年的发展，比特币程序已经出现了多个版本，比如 Bitcoin Classic、Bitcoin XT 以及 Bitcoin Unlimited，这些不同的版本实际上都是比特币核心程序的分叉版本，本节使用的是比特币核心程序的客户端。

现在先安装一个比特币核心客户端，按照下载地址进入页面后，在这个页面可以看到针对不同操作系统的下载版本，读者朋友可以自行选择，无论哪个系统环境下，其功能都是一样的，见下图：



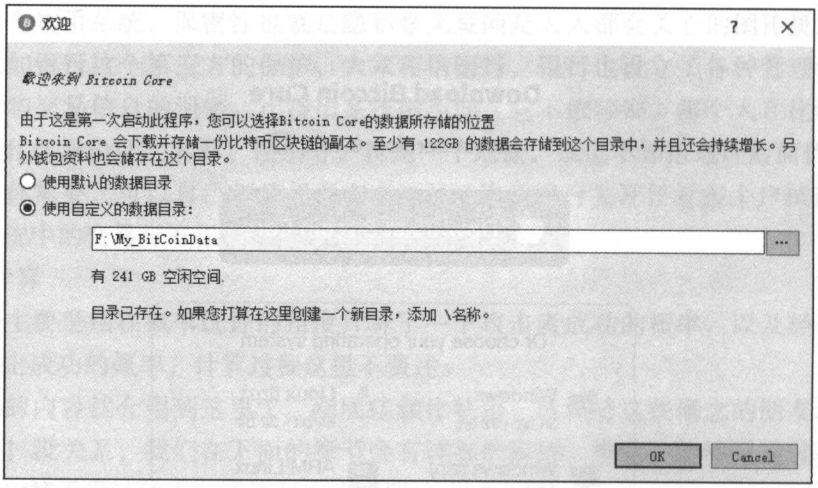
我们以 Windows 版本为例来说明，我们下载图中所示的 0.14.1 版。大家注意到没有，比特币发展了这么多年，到现在程序都还没进化到 1.0 版（通常一个软件的 1.0 版是首个正式版本），某种程度上也是因为比特币是一种实验性的软件吧，因此大家研究学习比特币可以带着一种玩的姿态，不要那么严肃，任何可能性都是有的，我们学习了解比特币是为了更好地应用它的设计思想，而不是去迷信它的神秘和权威。下载完成后，打开软件目录，可以看到有一个 bin 文件夹，其中有 5 个文件，如下图所示。



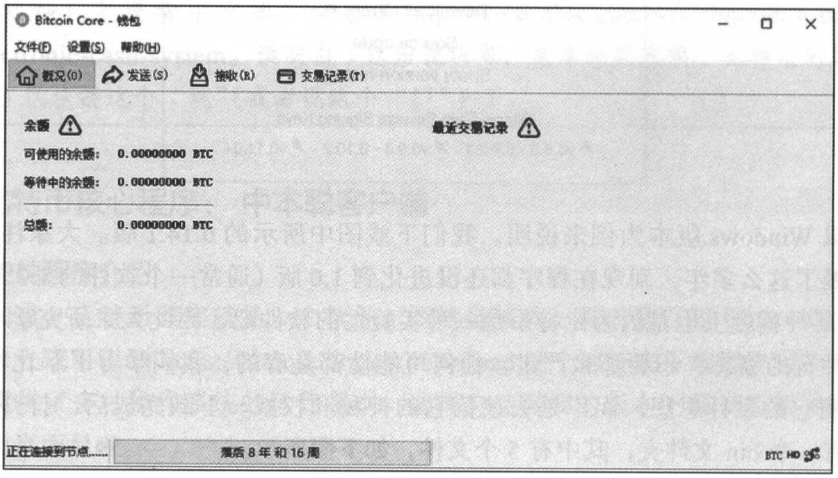
我们一一来说明一下：

（1）bitcoin-qt.exe

包含了比特币的核心节点以及一个钱包的前端功能，这是一个带有图形界面的客户端程序，运行后可以看到有如下提示：

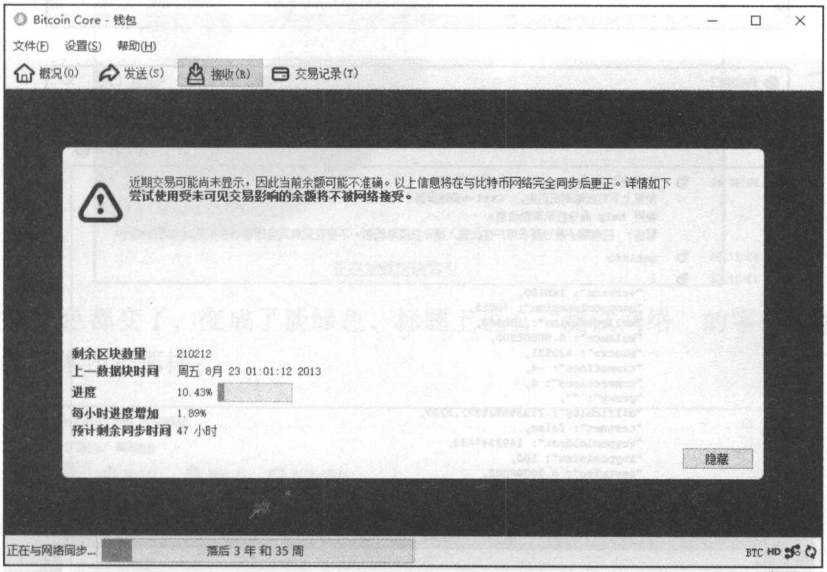


按图中所示，需要选择一个比特币的区块链副本数据存储目录，目前整个区块链账本数据已经很大了，选择一个空间足够大的目录，然后点击 OK 按钮即可进入主界面了，我们看下主界面的样子：



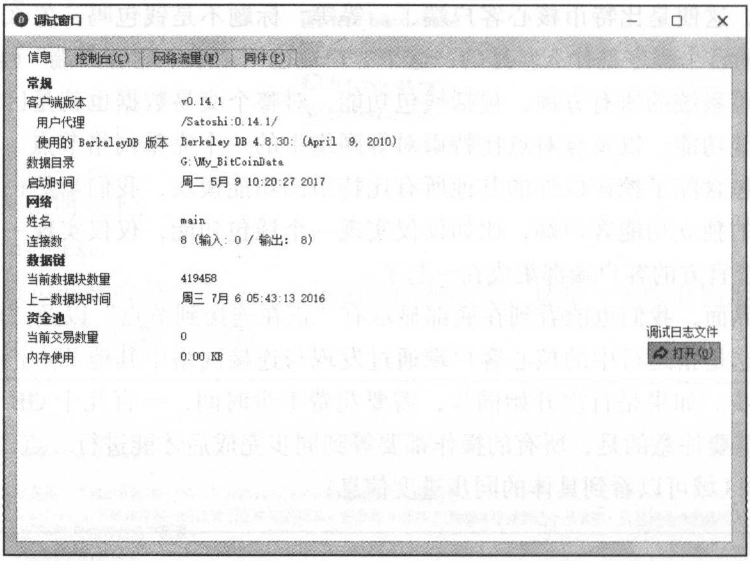
如图所示，这便是比特币核心客户端了。等等，标题不是钱包吗，怎么又是核心客户端，到底运行的这个程序是什么？是的，这个客户端也叫“中本聪客户端”（satoshi client），它实现了比特币系统的所有方面，包括钱包功能，对整个交易数据也就是区块链账本完整副本的交易确认功能，以及点对点比特币对等网络中的一个完整网络节点。换句话说，这个客户端软件包含除了挖矿以外的其他所有比特币的功能模块，我们当然也可以分别去自己实现一个个的独立功能客户端，比如仅仅实现一个钱包功能，仅仅实现一个核心节点功能，只不过这个官方的客户端都集成在一起了。

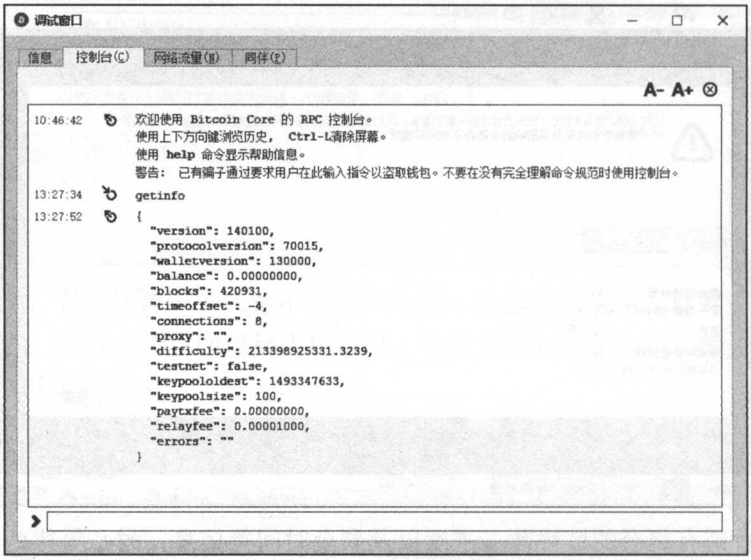
通过这个界面，我们也能看到在底部显示有“正在连接到节点”以及“落后 8 年和 16 周”的字样，这是指运行中的核心客户端通过发现与连接网络中其他节点进行区块链账本数据的一致同步。如果是首次开始同步，需要花费不少时间，一百几十 GB 的数据下载真够喝一壶的。需要注意的是，所有的操作都要等到同步完成后才能进行。点击那个“落后 8 年和 16 周”的区域可以看到具体的同步进度信息：



图中可以看到有剩余的区块数、进度以及剩余时间等信息，耐心等待就是了。如果想查看一下当前客户端的版本以及网络连接等信息，可以点击“帮助”→“调试窗口”调出如下界面。

在“信息”标签页下可以看到软件版本、当前的网络连接数、数据目录等摘要信息。注意这里的“客户端版本”，比特币是一个分布式的点对点系统，不存在中心服务器来统一管理软件的版本升级，因此不同的节点有可能运行着不同版本的客户端，不同版本的客户端在一些功能支持上可能会有些差异，大家在操作时一定要注意自己的版本。在“信息”标签页旁边有个“控制台”，这可是个很有用的功能，在控制台可以通过命令来访问核心客户端，调取一些信息，进行一些操作，我们来看下控制台。





我们在控制台底部的输入框中输入了一个 getinfo 命令，回车确认后可以发现返回了一段信息，这是关于当前运行的核心客户端节点的一些摘要信息，比如 version 表示核心客户端版本，protocolversion 表示协议版本，walletversion 表示钱包版本，balance 表示当前钱包中的比特币余额等。通过这个我们发现，比特币的核心客户端其实是充当了一个服务器的角色，通过控制台可以连接访问，通过界面也能看到提示："欢迎使用 Bitcoin Core 的 RPC 控制台。"实际上比特币核心客户端就是在启动的同时启动了一个本地的 RPC 服务，以方便外部程序进行相应的数据操作和访问。

有朋友问，比特币一下子要同步这么多的数据，而我只是想看一看，有没有试用的版

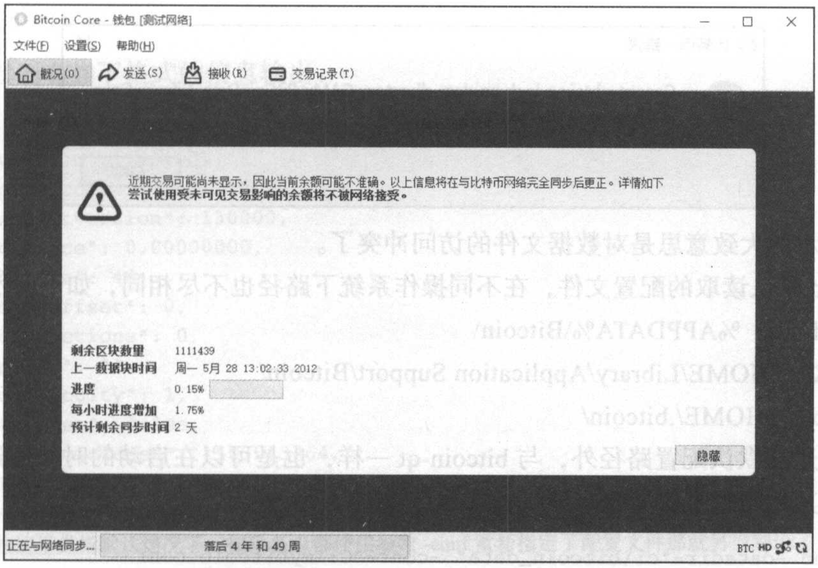
本呢？还真有，不过不叫试用版，而是测试网络。那么如何连接到测试网络呢？可以通过配置文件来进行配置。比特币的配置文件名为 bitcoin.conf，可以在数据目录也就是钱包数据文件 wallet.dat 所在目录下创建一个文本文件，命名为 bitcoin.conf 即可，这就是 bitcoin-qt 默认读取的配置文件了。接下来我们就来配置一下以进入测试网络，只需在 bitcoin.conf 中写入如下配置项：

```
testnet=1
```

保存即可，然后重新启动 bitcoin-qt.exe，我们可见如下画面：



我们发现颜色都变了，变成了淡绿色，标题上也有“测试网络”的字样，进入主界面后，界面样式基本还是那样：



进入到测试网络后的比特币客户端，其区块链数据会小一些，在功能操作上基本还是一样的。需要注意的是，配置文件中的配置项也是可以直接通过参数来传递的。假设想临时进入测试网络看看，那么就不需要去设置配置文件了，通过如下指令来运行即可：

```
bitcoin-qt -testnet
```

在控制台中执行上述指令后，同样会进入测试网络。有朋友会问，我一开始在运行 bitcoin-qt 时指定了一个数据目录，现在我想更换可以吗？当然是可以的，操作如下：

```
bitcoin-qt -datadir="D:\mybitcoin_data"
```

这样在启动 bitcoin-qt 的时候重新指定了一个自己创建的数据目录。当然了，不但可以重新指定数据目录，也可以重新指定配置文件，操作如下：

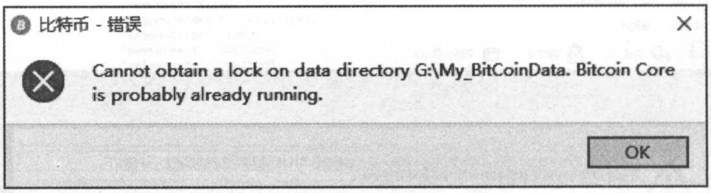
```
bitcoin-qt -conf="c:\mybitcoin.conf"
```

可以发现，另外指定的配置文件，其文件名可以是自定义的。需要注意的是，bitcoin-qt 支持的所有参数中，除了 -datadir 与 -conf 只能通过命令参数指定外，其他参数都是既可以在命令参数中直接传递，也可以在配置文件中指定。

（2）bitcoind.exe

这个其实就可以看作不带界面的 bitcoin-qt.exe，其中同样包含了比特币的核心节点，并且提供了 RPC 服务。比特币使用的是 JSON-RPC 协议，以便通过命令行交互的方式访问比特币系统的数据，比如访问区块链账本数据，进行钱包操作和系统管理等。

bitcoin-qt 与 bitcoind 是互相兼容的，有同样的命令行参数，读取相同格式的配置文件，也读写相同的数据文件，使用的时候，这两个程序根据需要启动一个即可，同时启动也不会出错，但是同时启动两个 bitcoin-qt 或者两个 bitcoind 会出错，如下所示：



图中所示的大致意思是对数据文件的访问冲突了。

bitcoind 默认读取的配置文件，在不同操作系统下路径也不尽相同，如下所示：

- Windows：%APPDATA%\Bitcoin\
- OS X：$HOME/Library/Application Support/Bitcoin/
- Linux：$HOME/.bitcoin/

除了上述的默认配置路径外，与 bitcoin-qt 一样，也是可以在启动的时候通过传递参数来重新指定其他路径下的配置文件或者数据目录的：

```
bitcoind -datadir="c:\bitcoin_data" -conf="C:\mybitcoin.conf"
```

如上所示，启动时，使用 -datadir 指定了数据文件需要存储的目录，使用 -conf 指定了 C 盘目录下的一个配置文件，此时这个配置文件的名称是自定义的。bitcoind 启动后可以通过 bitcoin-cli 进行访问，bitcoin-cli 的使用在下一节介绍。

看到这里，有些朋友可能会有些疑问，比特币核心客户端运行后可以与其他节点进行互相的连接通信，那就得开放一个服务端口，而访问比特币节点信息又是通过 RPC 的方式，那相当于开启了一个 RPC 服务，这么说来，比特币网络中的每个节点其实相当于一个个服务器。确实如此，这些开启的服务端口说明如下。

- 8333，用于与其他节点进行通信的监听端口，节点之间的通信是通过 bitcoin protocol 进行的，通过这个端口才能进入比特币的 P2P 网络。
- 8332，这是提供 JSON-RPC 通信的端口，通过这个端口可以访问节点的数据。
- 如果是测试网络，分别是 18333 和 18332。

以上端口是可以另外指定的，通过参数 -port 与 -rpcport 参数可以分别重新指定。

（3）bitcoin-cli.exe

bitcoin-cli 允许你通过命令行发送 RPC 命令到 bitcoind 进行操作，比如 bitcoin-cli help，因此这是一个命令行客户端，用来通过 RPC 方式访问 bitcoind 的 RPC 服务。我们可以通过命令行来查看当前的 bitcoin-cli 的版本：

```
bitcoin-cli -version
```

运行后会返回如下描述信息：Bitcoin Core RPC client version v0.14.2。通过返回的信息也能看到，bitcoin-cli 就是一个 RPC 客户端工具，那么如何去连接核心客户端呢？首先 bitcoin-cli 与 bitcoind 是使用同样路径下的配置文件[⊖]，因此在使用 bitcoin-cli 之前，我们需要先运行 bitcoind，然后来执行 bitcoin-cli 命令：

```
bitcoin-cli getinfo
```

可以看到有如下格式的信息输出：

```
{
    "version": 140100,
    "protocolversion": 70015,
    "walletversion": 130000,
    "balance": 0.00000000,
    "blocks": 48,
    "timeoffset": 0,
    "connections": 0,
    "proxy": "",
    "difficulty": 1,
    "testnet": false,
    "keypoololdest": 1503043764,
    "keypoolsize": 100,
```

⊖ 当然这里是指默认情况下，如果各自都使用参数 -conf 重新指定了配置文件那就另当别论了。

```
    "paytxfee": 0.00000000,
    "relayfee": 0.00001000,
    "errors": ""
}
```

看到信息的返回，表明已经正常连接且可以访问了，如果想要停止 bitcoind，则可以发送如下指令：

```
bitcoin-cli stop
```

bitcoind 接收到停止命令，执行后退出运行服务。

我们再来看一个例子，在这个例子中，通过参数重新指定数据目录和配置文件：

```
bitcoind -datadir="c:\bitcoin_data" -conf="C:\bitcoin.conf"
```

此时，如果仍然要通过 bitcoin-cli 来访问这个运行的 bitcoind，则需要运行如下命令：

```
bitcoin-cli -datadir="c:\bitcoin_data" -conf="c:\bitcoin.conf" getinfo
```

运行后返回了运行的 bitcoind 中的信息。

至此，我们可以发现，bitcoin-qt、bitcoind 以及 bitcoin-cli 都能读取相同格式的配置文件，也拥有一样的命令参数。具体支持的各种参数很多，大家可以自行去查阅。另外，比特币中的很多功能调用都是通过 RPC 命令提供的，比如区块信息查询、交易事务查询、多重签名使用等，因此要了解完整功能调用的朋友可以去具体了解一下这些 RPC 命令的使用，笔者这里也推荐一些不错的网站方便大家学习使用：

- https://blockchain.info：方便检索各项比特币网络的数据；
- https://chainquery.com/bitcoin-api：基于网页的比特币 RPC 命令使用。

（4）bticoin-tx.exe

这是一个独立的工具程序，可以用来创建、解析以及编辑比特币中的交易事务。我们在通常使用比特币系统的时候，使用上述介绍的钱包功能也就足够了，但是如果需要单独查看或者创建一份交易事务数据，就可以使用这个工具了。既然是用于操作交易事务的，那么我们就来试一试。比特币的交易事务在本质上就是一段二进制数据，我们任意找一段过来，看看 bitcoin-tx 能解析成什么样子，为了方便，将二进制的交易事务数据转成十六进制的格式来显示，如下：

```
0100000001e0772cd81114d0993922a280e2b29209d6c6c5d2f22d807018d1ef0d55cfe4041c0
000006a47304402200865Ob496ea573a2d42efbcbfb49288ab3c7f9968a1fa6072155a028a4de
b39e02201b2dd03307fcd1fbb2f9928a8904d50a84ae9d600986a3a8a125fe248b4faf1001210
354eb6c85025f3abecde8236e86aabf6b819a72154e69d39f7ae591a92436c166ffffffff01d9
38890c000000001976a914fe5d8413d80c3d3f9b975f45990cf432455b13ef88ac00000000
```

这就是一段交易事务的数据，接下来我们就来解析一下，将这段数据转换成容易阅读的格式，为了方便阅读，我们就转换为 JSON 格式，命令如下：

```
bitcoin-tx -json
```

0100000001e0772cd81114d0993922a280e2b29209d6c6c5d2f22d807018d1ef0d55cfe4041c0
000006a47304402200865 0b496ea573a2d42efbcbfb49288ab3c7f9968a1fa6072155a028a4de
b39e02201b2dd03307fcd1fbb2f9928a8904d50a84ae9d600986a3a8a125fe248b4faf1001210
354eb6c85025f3abecde8236e86aabf6b819a72154e69d39f7ae591a92436c166ffffffff01d9
38890c000000001976a914fe5d8413d80c3d3f9b975f45990cf432455b13ef88ac00000000

执行后，可以得到如下的输出：


```
{
    "txid": "2aff308e3e1a9b251ecb701762f6f2c1d28952fe6d0d94efc78880e8a62d2cbb",
    "hash": "2aff308e3e1a9b251ecb701762f6f2c1d28952fe6d0d94efc78880e8a62d2cbb",
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "txid":
"04e4cf550defd11870802df2d2c5c6d60992b2e280a2223999d01411d82c77e0",
            "vout": 28,
            "scriptSig": {
                "asm": "3044022008650b496ea573a2d42efbcbfb49288ab3c7f9968a1fa607
2155a028a4deb39e02201b2dd03307fcd1fbb2f9928a8904d50a84ae9d600986a3a8a125fe248b4faf10[
ALL] 0354eb6c85025f3abecde8236e86aabf6b819a72154e69d39f7ae591a92436c166",
                "hex": "473044022008650b496ea573a2d42efbcbfb49288ab3c7f9968a1fa
6072155a028a4deb39e02201b2dd03307fcd1fbb2f9928a8904d50a84ae9d600986a3a8a125fe248b4f
af1001210354eb6c85025f3abecde8236e86aabf6b819a72154e69d39f7ae591a92436c166"
            },
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 2.10319577,
            "n": 0,
            "scriptPubKey": {
                "asm": "OP_DUP OP_HASH160 fe5d8413d80c3d3f9b975f45990cf432455b13
ef OP_EQUALVERIFY OP_CHECKSIG",
                "hex": "76a914fe5d8413d80c3d3f9b975f45990cf432455b13ef88ac",
                "reqSigs": 1,
                "type": "pubkeyhash",
                "addresses": [
                    "1QBxfKsz2F7xwd66TwMj5wEoLxCQghy54c"
                ]
            }
        }
    ],
    "hex": "0100000001e0772cd81114d0993922a280e2b29209d6c6c5d2f22d807018d1ef0d55
cfe4041c0000006a473044022008650b496ea573a2d42efbcbfb49288ab3c7f9968a1fa6072155a028a4d
eb39e02201b2dd03307fcd1fbb2f9928a8904d50a84ae9d600986a3a8a125fe248b4faf1001210354eb6c
85025f3abecde8236e86aabf6b819a72154e69d39f7ae591a92436c166ffffffff01d938890c00000000
1976a914fe5d8413d80c3d3f9b975f45990cf432455b13ef88ac00000000"
}
```

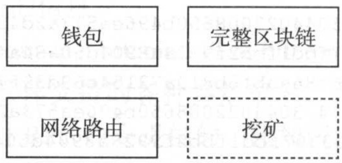
通过输出信息，我们可以很方便地看到其中包含的各个数据组成项，比如 txid 是指交易事务的哈希值，这个值与哈希数据项一样；vin 是指交易事务中的输入部分；vout 是指交易事务中的输出部分，具体每一项的含义这里暂且不多解释，第 8 章通过模拟比特币构建一个最简易的区块链系统，其中有具体的介绍。通过使用这个工具，除了能解析交易事务数据外，也能创建交易事务，读者朋友们可以去具体尝试一下。

（5）test_bitcoin.exe

这是用于比特币程序 bitcoind 的单元测试工具，与程序开发相关，除了这个，实际上还有一个用于 bitcoin-qt 的单元测试工具 test_bitcoin-qt，这些工具普通用户一般用不到，这里不再展开详述。

2. 客户端逻辑结构

通过上述介绍，我们了解了中本聪客户端程序的基本组成，为了让大家有一个更加清晰的理解，我们来看下中本聪客户端在逻辑结构上包含了哪些功能模块，见下图：



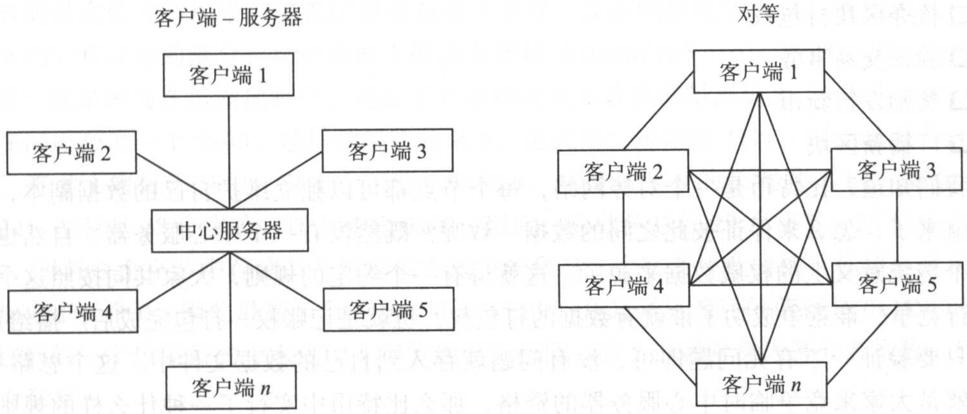
图中所示的 4 个功能模块，共同组成了称为全节点的比特币程序结构，其中“挖矿”部分标记了虚线，这是因为在中本聪客户端中没有包含挖矿功能，挖矿是另外独立的程序。钱包的功能我们已经比较了解了，主要用于管理用户的密钥以及提供转账操作等功能，属于比特币的前端功能。事实上，钱包功能是可以独立出来的，专门提供一个独立的钱包程序，这部分的阐述在下面章节中有详细描述。接下来，我们主要对“完整区块链”和“网络路由”部分进行说明。刚才说了，钱包只是一个前端功能，那么比特币的后端功能是什么呢？请看下文。

（1）完整区块链

中本聪客户端保留了完整的区块链账本数据，因此能够独立自主地校验所有交易，而不需借由任何外部的调用。当然，另外一些节点只保留了区块链的一部分（比如区块头），可以通过一种名为“简易支付验证”（SPV）的方式来完成支付验证，这样的节点被称为“SPV 节点”。除了中本聪客户端外，一些挖矿节点也保有了区块链的完整数据副本，还有一些参与矿池挖矿的节点是轻量级节点，它们必须依赖矿池服务器维护的全节点进行工作。保有完整区块链数据的节点是非常重要的，比特币网络之所以能够成为一个可信任的去中心化网络，就是依赖于这些全节点，目前很多场合为了方便使用，提供了不少轻量级节点（如轻钱包等），但是这些轻量级节点的正常使用都是要通过全节点才能完成的，是一种依赖关系，如果网络中保有完整区块链数据的节点越来越少，那么比特币网络就会受到影响，无论性能、安全性等都会降低。

（2）网络路由

比特币网络是属于 P2P 网络架构，P2P 也就是对等的意思，与此相对的是“客户端 - 服务器”架构，有一个提供服务功能的中心服务器，其他客户端通过调用服务器的功能来完成操作，比如我们通常使用的微信、支付宝、网银等，如果提供商的服务器关闭了，那也就完全没法使用这些软件了。在对等网中，每个节点共同提供网络服务，不存在任何所谓的中心服务器，因此在对等网络的网络架构中是没有层次的，大家都是平等的，每个节点在对外提供服务的同时也在使用网络中其他节点所提供的服务。我们来看下两者的区别示意图：



一目了然了，在“客户端 - 服务器”网络架构中，总是有一个中心的，一旦中心服务器出了问题，基本等于天塌了；而“对等”网络结构，相比中心化服务器这种单点故障结构有很强的抵抗能力，我们可以看到，“对等”结构中的节点都是可以与其他节点互连的，而且某个节点出问题也不影响其他节点之间通信，这种结构的好处显而易见。当然，无论哪种网络结构，底层的网络协议都是一样的，还是 TCP/IP 那一套。

比特币是属于区块链技术的首创应用，其特点就是去中心化或者说是分布式，由比特币节点组成的网络自然也就是属于“对等”网络了，那么既然没有一个服务器，大家彼此如何来认识对方呢，即如何发现其他的节点呢？这是需要通过一个协议的，首先节点会启动一个网络端口[⊖]，通过这个网络端口与其他已知的节点建立连接。连接时，会发送一条包含认证内容的消息进行“握手”确认，比特币网络中是靠彼此共享节点信息来寻找其他节点的，当一个节点建立与其他节点的连接后，会发送一条包含自身 IP 地址的消息给相邻的节点，而邻居收到后会再次发送给自己的其他邻居，当然节点也不是只能被动地等别人来告诉自己，也可以自己发送请求给其他节点索取这些地址信息，如果与发现的节点之间能够成功连接，那么就会被记录下来，下次启动时就会自动去寻找上次成功连接过的节点。

简单地说，作为网络路由的功能，比特币节点在失去已有连接时会去发现新节点，同

⊖ 通常是 8333，但也可以参数指定，在 1.4.2 节中介绍中本聪客户端时已经说明过。

时自己也为其他节点提供连接信息，没有服务器的对等网络就是这么来认识陌生人的。

至此，大家对比特币的核心客户端就有了一个较为完整的理解了吧。

1.4.3 比特币的发行：挖矿

很多朋友在第一次看到“挖矿”这个词时都很疑惑，包括本人。比特币不是一个软件吗？通过软件来挖矿是什么意思？从字面上来看，应当是通过投入某种工作，然后能得到一个“宝贝”，也就是矿。当然了，“挖矿”自然不是我们通常认为的那个挖矿，它只是一套算法，在介绍算法过程前，我们先来了解下挖矿在比特币软件中主要都有哪些用途：

- 抢夺区块打包权
- 验证交易事务
- 奖励发行新币
- 广播新区块

我们知道，比特币是一个对等网络，每个节点都可以独立维护自己的数据副本，那么问题就来了，怎么来保证彼此之间的数据一致呢？既然没有一个中心服务器，自然也就没有一个传统意义上的权威数据来源了。这就得有一个约定的规则，大家共同按照这个规则来进行竞争，谁竞争成功了谁就有数据的打包权，也就是记账权，打包完成后广播给别人，别人只要验证一下有无问题即可，没有问题就存入到自己的数据文件中。这个思路不错，等于就是大家来竞争临时中心服务器的资格，那么比特币中实行了一种什么样的规则呢？那就是被称为工作量证明（Proof of Work，PoW）的一种算法，其实就是类似于掷骰子的一种游戏。比如大家约定掷出一个10位长度的数字，前面6位要都是0，后面的4位数得小于某个值，看谁先掷出符合要求的数字出来，谁就抢得了打包权（记账权）。我们来看下比特币中具体是怎么来掷这个骰子的。

1. 难度值

首先，既然是大家都在竞争掷骰子，那掷出来的数字必然是要符合一个难度的，这个难度就是一个门槛，在比特币软件中，规定一个256位的整数：

x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

作为难度1的目标值。在比特币诞生初期，当时的全网算力，大约需要10分钟左右的运算能得到一个符合这个难度1要求的值，这也是我们常常说比特币网络每隔大约10分钟出一个区块的来源。我们在查询创世区块（也就是0号区块）的信息时，可以看到当时的难度就是1。那么，所谓符合这个难度为1的要求的值是什么意思呢？就是说通过工作量证明算法，也就是比特币中的挖矿算法来计算出一个结果，这个结果要小于这个难度目标值，我们来看下0号区块的难度信息：

```
"nonce": 2083236893,
"bits": "1d00ffff",
```

```
"difficulty": 1,
```

这些信息可以通过比特币支持的 JSON-RPC 中的 getblock 命令方法获得，其中的 difficulty 就是指难度级别。0 号区块的难度值是 1，nonce 是一个随机数，是挖矿计算得到的一个数字，这个等会儿再介绍，bits 是用来存储难度的十六进制目标值的，这个难度目标值是存储在区块的头部的，在源码中被定义为一个 4 字节长度的字段，4 字节也就是 32 位，要用来存储 256 位长度的难度目标值，因此这 256 位长度的值需要经过压缩处理后才能放到这个字段中。以这个难度 1 的目标值来说，我们查询区块信息后，看到的值是 1d00ffff，那么，这个值是怎么压缩而来的呢？规则其实很简单，我们一共有 4 个字节来存储，这 4 个字节的最高位字节用来存储难度值的有效字节数。什么叫有效字节数？就是从第一个不全为 0 的字节开始的部分，比如难度 1 的值有效位是 0x00FFFF……。等等，怎么前面有 2 个 0 呢？这是因为在压缩规则中，规定了如果难度值有效位的最高位为 1（大于 0x80），则需要在前面补上一个 0x00，这里的最高位是 F，也就是二进制的 1111，因此是符合这个规则的。难度 1 的目标值中，有 4 个字节长度的 0，减掉这些 0 的长度共 32bit，剩余 256-32=224，也就是 28 个字节，加上补的 0x00，因此，有效位总计 29 个字节，29 的十六进制是 1D，另外 3 个字节中存储的是目标值有效位的最高 3 个字节，此时的目标值有效位前面已经加上了 2 个 0，因此最高 3 个字节为 0x00FFFF，合起来压缩后的值就是 0x1D00FFFF。对于这样的一个压缩后的十六进制 4 字节难度目标值，前 2 位通常称为幂或者指数，后面 6 位称为系数。

那么，有朋友问了，压缩是可以，那还原出来呢？我们看个公式：目标值 = 系数 * 2^(8*(指数 -3)) 次方。

我们就以这个 0x1D00FFFF 为例来说明，系数是后面 6 位也就是 00FFFF，指数是前面 2 位也就是 1D，代入进去就是：0x00FFFF*2^(8*(0x1D-3))，计算后得到的值是：

0x00000000FFFF0000000000000000000000000000000000000000000000000000

有朋友可能疑惑了，不对吧，这个跟规定的那个难度 1 的值不一样了啊，精度少了很多，确实是的，存储在 bits 中的值是一个精度截断的近似值。

我们以 200 000 号区块为例，查询一下难度值，得到如下：

```
"nonce": 4158183488,
"bits": "1a05db8b",
"difficulty": 2864140.507810974,
```

我们来看看这个 difficulty 的值是怎么来的，0 号区块的难度是 1，对应的目标值是 0x1D00FFFF，200 000 号区块的难度目标值是 0x1A05DB8B，将两者的目标值按照上述公式进行转换后相除便能得到这个 2864140.507810974 的难度值，我们发现，200 000 号区块的 difficulty 比 0 号区块的大许多，而 bits 的大小却比 0 号区块的小许多。这其实是表明了一个特点，随着全网算力越来越强，difficulty 难度值就会越来越大，而 bits 表示的目标值会越来越小，这两者成反比，目标值越小就越难挖矿。

刚才也提到了，难度值并不是一成不变的，比特币差不多每两周会调整一下新的难度值，因为计算的算力是会变化的，为了维持差不多10分钟出一个区块的节奏，难度要跟随算力变化而调整，不得不说比特币的设计还是相当完整的。

新难度值的计算公式是这样的：新难度值 = 当前难度值 ×（最近的2016个区块的实际出块时间 / 20 160 分钟）。2016个区块的意思是：假设按照理论的10分钟出一个块，2周也就是14天的时间，应该出2016个区块，可以看到实际上就是计算一下实际与理论上的时间差值，弥补上这个差值即可。

2. 挖矿计算

我们了解了难度值的概念，现在来看看挖矿计算具体是怎样一个过程。首先，我们说了挖矿是要抢夺区块打包权，那就得收集需要打包进区块的那些交易事务，那这些数据从哪来呢？这里有个概念需要大家注意，打包就像是记账，是把发生的交易事务记录下来存档，但是无论什么时候打包、谁打包，在网络中发生的交易是持续不断的，就像企业仓库的进销存业务，无论记账员是一个月还是半个月记一次账，业务是持续进行的。在比特币系统中，每个人都会将通过钱包进行的转账交易数据广播到网络中，这些都是属于等待打包的未确认交易数据。这些数据都会放在一个内存池中，总之就是一个缓冲区，当然，这些数据都会被接受基本的验证，用以判断是否是不合法的或者是不符合格式的交易数据。

挖矿程序从内存池中获取用来打包区块的交易数据，接下来就要干活啦，我们来看一下挖矿的计算公式：

```
SHA256(
        SHA256(version + prev_hash + merkle_root + ntime + nbits + nonce )
) < TARGET
```

SHA256是一种哈希算法，可以通过对一段数据进行计算后输出一个长度为256位的摘要信息。SHA256在比特币中使用很广泛，不但用于挖矿计算，也用于计算区块的哈希值和交易事务的哈希值，比特币对SHA256算法是情有独钟啊，我们看到在这个公式中，是对参数进行两次SHA256计算，如果计算出来的值小于那个TARGET（也就是难度目标值），那就算是挖矿成功了。那么，这些参数都是由哪些组成的呢？请看下表：

名称	含义
version	区块的版本号
prev_hash	前一个区块的哈希值
merkle_root	准备打包的交易事务哈希树的根值，也就是梅克尔根
ntime	区块时间戳
nbits	当前难度
nonce	随机数

这些数据字段其实也是区块头的组成部分，将这些参数连接起来，参与SHA256的挖

矿计算。在这些参数中，版本号是固定的值，前一个区块的哈希值也是固定的值，当前难度也是一个固定的值，那么要想改变这个公式的计算结果，能改动的参数就只有梅克尔根、区块时间戳和那个随机数了。

1）梅克尔根是通过交易事务计算出来的，挖矿程序从内存池中获取待打包的交易事务，然后计算出梅克尔根，获取交易事务本身也是有一些优先级规则的，比如根据手续费大小之类，这些细节就不赘述了。

2）区块时间戳是指 UNIX 时间戳，用于记录区块的产生时间，我们知道比特币系统是分布式的网络，没有固定的时间服务器，因此每个节点获得的时间戳都可能是不一样的，由此，比特币系统中设置了规则：①新产生区块的时间戳要大于之前 11 个区块的平均时间戳；②不超过当前网络时间 2 个小时。所以，后一个区块的时间戳比前一个区块的时间戳反而小也是可能的。

3）随机数是一个可自由取值的数值，取值范围是 0 ～ 2 的 32 次方。

我们可以看到，要通过这样的参数来计算出符合条件的值，基本上也就只能靠暴力计算匹配了，这种不断执行 SHA256 计算的过程很消耗算力，因此这个过程被形象地称为“挖矿”。简单地说，**挖矿就是重复计算区块头的哈希值，不断修改该参数，直到与难度目标值匹配的一个过程。**

一旦匹配成功，就可以广播一个新的区块，其他客户端会验证接收到的新区块是否合法，如果验证通过，就会写入到自己的区块链账本数据中。那么，挖矿的奖励在哪儿呢，不是说矿工成功出一个区块就能得到比特币作为奖励的吗？那么这里奖励在哪呢？这个奖励其实是作为一条交易事务包含在区块的交易事务中的，相当于系统给矿工转账了一笔比特币，这种交易事务由于特殊性，通常称为 coinbase 交易，这个交易一般是位于区块中的第一条，比特币系统也正是通过这种挖矿奖励的方式发行新的比特币，就像央行发行新钞一样。

这个奖励不是无限的，从 2009 年 1 月创建出第一个区块，每个区块奖励 50 个比特币，然后每 21 万个区块（大约 4 年）产量减半，到 2012 年 11 月减半为每个区块奖励 25 个比特币，然后在 2016 年 7 月减半为每个新区块奖励 12.5 个比特币。基于这个公式，比特币挖矿奖励逐步减少，直到 2140 年，所有的比特币（20 999 999.98）将全部发行完毕，到那个时候挖矿就只能收入一些交易手续费了。彼时，比特币网络是否还能保持运行，我们目前也只能持保留意见了。矿工在没有明显的激励情况下，是否还愿意通过挖矿承担区块打包的责任，现在也很难说。

比特币中的挖矿计算基本就是这个过程了，其实还是很简单的，本质上就是利用了 SHA256 计算，有朋友可能有疑问，那第一个区块也就是创世区块是怎么挖出来的？很简单，创世区块是硬编码直接写进去的，在比特币的源码中，通过 CreateGenesisBlock 这个方法写入，并且还留下了一句话：The Times 03/Jan/2009 Chancellor on brink of second bailout for banks。当时正是英国的财政大臣达林被迫考虑第二次出手缓解银行危机的时刻，

这句话是泰晤士报当天的头版文章标题。

3. 区块广播

矿工挖出区块后，就进行网络广播，传递给相邻的节点，节点接收到新的区块后会进行一系列的验证，比如区块数据格式是否正确；区块头的哈希值小于目标难度；区块时间戳是否在允许范围之内；区块中第一个交易（且只有第一个）是 coinbase 交易；区块中的交易事务是否有效等，总之就是一连串的检测，全部校验通过就把新的区块数据纳入到自己的区块链账本中。如果是挖矿节点接收到信息，就会立即停止当前的挖矿计算，转而进行下一区块的竞争。

比特币的挖矿过程说到这里，不知道有没有朋友会有个疑惑，那就是挖矿算法虽然能够提供工作量证明，表明矿工确实是投入了相当的算力的，但是却不能保证只能是一个矿工能挖到啊，如果在同一时间内多个矿工都计算出了符合条件的值，都拥有了打包权，那以谁的为准呢？比特币中的解决方案，竟然是那么简单，人家没用什么复杂的算法，就是让节点自己选择，最终传播最广、处于最长链中的区块将被保留，因此到底谁的区块会被保留下来，可能还真得看看运气了。

这里实际上隐含着 FLP 原理，先看下定义：在网络可靠，存在节点失效（即使只有一个）的最小化异步模型系统中，不存在一个可以解决一致性问题的确定性算法。这个其实也很好理解，来看个例子：三个人在不同房间投票，虽然三个人彼此之间是可以通电话沟通的，但是经常会有人时不时地睡着。比如，A 投票 0，B 投票 1，C 收到了然后睡着了（类比节点失效了），则 A 和 B 永远无法在有限时间内和 C 共同获得最终的结果。看到这里，我们也就明白了挖矿的作用了，除了发行新的比特币外，主要就是维持网络共识，让每个节点对区块链的数据保持最终一致性。

4. 挖矿方式

比特币的挖矿过程我们已经了解了，现在给大家介绍下挖矿方式。挖矿算法在执行过程中，为了抢夺区块打包权，就得拼命去算出那个符合难度目标的值，大家都在不断升级自己的算力，难度也就越来越大，挖矿程序本身不复杂，关键是这个过程非常依赖计算机的算力资源，可以说得算力者得天下，也因为这个原因，挖矿的方式在多年来不断进化，一切都围绕着为了得到更高的算力来进化。

我们先从硬件类型来说，早期的时候，还没多少人挖矿，难度值也还不大，使用普通的个人电脑就能进行挖矿了，这个时期的硬件设施主要是普通 CPU 挖矿。随着有更多的矿工加入，难度越来越大，使用普通 CPU 的算力，效率开始不够用，于是出现了 GPU 挖矿，利用显卡来进行挖矿计算，GPU 对于 SHA256 的计算性能更高。曾经一段时间，市面上的显卡销量猛增，就是被买去搭建显卡挖矿的，2017 年上半年，另外一种数字加密货币以太坊价格暴涨，也一度引发了市面上的“一卡难求”，显卡尤其是高端显卡的 GPU 计算在一

些挖矿算法上的性能表现确实相当不错，然而，对于算力的追求是无止境的，接着又出现了配置 FPGA（Field-Programmable Gate Array，现场可编程门阵列）和 ASIC（Application Specific Integrated Circuit，特定应用集成电路）的挖矿装备。这两类是属于集成电路的装备了，尤其是 ASIC，基本是目前顶级性能的矿机了，专门为了挖矿而设计，只为挖矿而生！

说完了挖矿的装备，我们再来说说挖矿节点的类型，最简单的挖矿节点类型就是 solo 挖矿，也就是个体矿工，自己搞个挖矿装备然后默默开挖，守株待兔般等待着挖矿成功。如今，在挖矿难度大幅度提升的时代，个人挖矿几乎是一点机会都没有，那么现在流行的挖矿节点是什么类型？那就是矿池，矿池通过挖矿协议协调众多的矿工，相当于大家联合起来，每个人都贡献自己的算力，形成一个整体，大大增强整个挖矿节点网络的算力，个人矿工也可以加入到矿池，他们的挖矿设备在挖矿时保持和矿池服务器的连接，和其他矿工共同分享挖矿任务，之后分享奖励。

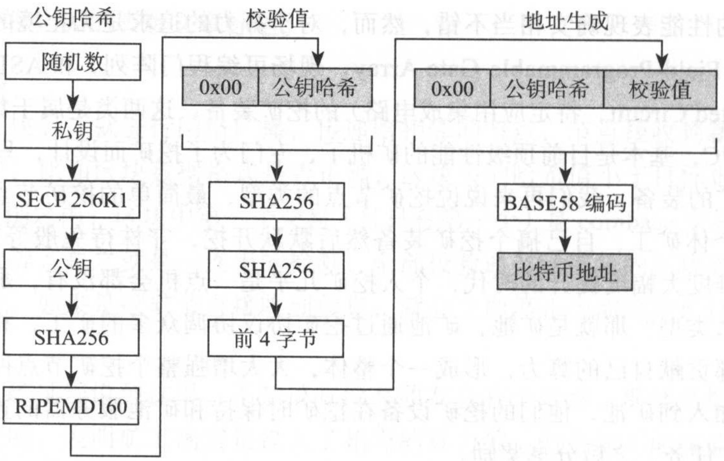
1.4.4　比特币钱包：核心钱包与轻钱包

钱包，是属于比特币系统中的一个前端工具，其最基本的功能就是用来管理用户的比特币地址、发起转账交易、查看交易记录等，在这方面与我们生活中使用的钱包是类似的。一开始的比特币钱包是跟比特币核心客户端一起发布的，1.4.2 节介绍比特币核心客户端的时候已经初步做了了解，这个钱包是比特币核心钱包，其使用过程必须要配合完整的区块链数据副本，因此一般也只适合在桌面端使用。

我们在使用比特币钱包的时候，经常会遇到一个名词：比特币地址。通过钱包转账就是将比特币从一个地址转移到另外一个地址，暂且不论这个转移的过程是什么样的，那这个地址到底是什么意思，它又是通过什么来产生的呢？我们先来看一组名词关键字：私钥、公钥和钱包地址。

私钥与公钥来自公开密钥算法的概念，我们常说比特币是一种加密数字货币，之所以这么说，是因为比特币的系统设计中巧妙地使用了现代加密算法，而其中一个运用就是生成比特币地址，比特币地址的生成与公开密钥算法密切相关。什么叫公开密钥算法呢？传统的加密算法，其加密和解密方法是对称的，比如凯撒密码，通过将字母移位来加密，比如字母 a 替换成 c，b 替换成 d，d 替换成 f 这样，本来是 abc 的单词就变成了 cdf，然而这种加密算法一旦泄露，别人也就知道了解密算法，换句话说，只有一个密钥。针对这种问题，公开密钥算法就应运而生，而公开密钥算法属于一种不对称加密算法，拥有两个密钥：一个是私钥，一个是公钥。公钥可以公开给别人看到，私钥必须要妥善保存，使用私钥加密（通常习惯上将私钥加密称为“私钥签名”）的数据可以用公钥解密，而使用公钥加密的数据可以用私钥解密，两者是互相匹配的。目前使用比较广泛的公开密钥算法主要有 RSA 算法和椭圆曲线加密算法（ECC），RSA 是利用了素数分解难度的原理，ECC 是利用了椭圆曲线离散对数的计算难度，比特币中使用的是椭圆曲线加密算法。

接下来，我们就来看下比特币地址是怎么生成的，为了直观展示，我们看一幅示意图：



这便是比特币地址的生成过程，过程大致是这样的。

1）首先使用随机数发生器生成一个私钥，私钥在比特币中的作用非常重要，可以用来证明用户的身份，也可以签发交易事务。

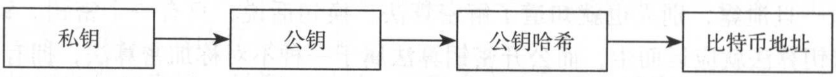
2）私钥经过 SECP256K1 算法处理生成了公钥，SECP256K1 是一种特定的椭圆曲线算法，需要注意的是，通过算法可以从私钥生成公钥，但是却无法反向从公钥生成私钥，这也是公钥为什么可以公开的原因。

3）公钥接下来先使用 SHA256 哈希算法计算，再使用 RIPEMD160 哈希算法计算，计算出公钥哈希。比特币的代码通过 2 次哈希来计算地址值，这样能进一步确保哈希后的数值唯一性，进一步降低不同数据进行哈希后相同的概率。与 SHA256 一样，RIPEMD160 也是一种哈希算法。

4）将一个地址版本号连接到公钥哈希（比特币主网版本号为 0x00），然后对其进行两次 SHA256 运算，将计算得到的结果取前面 4 字节作为公钥哈希的校验值。

5）将 0x00 版本号与公钥哈希以及校验值连接起来，然后进行 BASE58 编码转换，最终得到了比特币地址。

以上便是比特币地址的生成过程了，我们可以发现比特币的地址其实就是通过公钥转化而来的，将上图简化一下，就是下面这么一个过程：



所以，在比特币系统中，本质上并没有一个叫作“地址”的东西，因为“地址”是可以通过公钥转化而来的，可以理解为公钥的另外一种形式，而公钥又是可以通过私钥计算出来的，因此在比特币钱包中，真正需要妥善保存的是生成的私钥数据，这玩意可千万不能弄丢了，一旦丢失，那可比忘记银行卡密码还麻烦。比特币钱包的主要功能就是保管私钥。

比特币的核心钱包是跟核心客户端在一起的，可以完成创建钱包地址、收发比特币、

加密钱包、备份钱包等功能，由于核心钱包是与核心客户端在一起使用的，因此在进行转账交易时，可以进行完整的交易验证，当然付出的代价就是必须得带上那么大量的账本数据，到 2017 年 8 月份这份数据已经超过了 130GB，而且还在持续不断地增长中，因此并不方便用户的实际使用，实际上除了这一点不方便外，在私钥管理上也有麻烦的地方，通过官方的核心钱包可以无限制地创建自己所需数量的钱包地址，然而这些地址对应的私钥管理也就成了问题，如果不小心损坏了某一个私钥数据，那就找不回来了，基于这些问题，发展出了新的解决方案。

很多时候，我们在进行支付的时候，只是想通过一个支付验证，知道支付已经成功发起就可以了。对于完整的交易验证（需要在完整的账本数据上校验，比如是否包含足够的余额，是否双花等）可以交给核心节点，这样就可以将钱包功能部分剥离出来，由此产生了 SPV 钱包，事实上这个概念在比特币白皮书中就介绍过了，我们来看下它的原理是什么，SPV 钱包的大致过程如下所示。

1）首先下载完整的区块头数据，注意是区块头，而不是所有的区块链数据，这样可以大大减少需要获取的账本数据量，区块头中包含有区块的梅克尔根，SPV 方式主要就是靠它来实现的。

2）如果想要验证某笔支付交易，则计算出这笔交易事务的哈希值 txHash。

3）找到 txHash 所在的区块，验证一下所在区块的区块头是否包含在账本数据中。

4）获得所在区块中计算梅克尔根所需要的哈希值。

5）计算出梅克尔根。

6）若计算结果与所在区块的梅克尔根相等，则支付交易是存在的。

7）根据该区块所处的高度位置，还可以确定该交易得到了多少个确认。

我们看到了，SPV 原理的钱包就是使用了梅克尔树来验证支付是否已经发生，这也是为什么称之为简单支付验证的原因，不过我们也可以发现，支付验证所做的事情很少，仅仅能看到当前的支付交易是否被发起而已，并不能保证这笔交易事务最终会进入到主链中，也就是说还需要等待核心节点进行全面的交易验证并且矿工打包到区块后进入主链。在这个过程中是有可能发生失败的，所以 SPV 钱包虽然带来了便捷性但也牺牲了安全性。时至今日，已经出现了各种各样的比特币钱包，在 bitcoin.org 网站上我们可以一见端倪：



我们可以看到有各种类型的钱包可以使用，大家在选用自己的钱包时，务必了解清楚钱包的功能和来源，以免遭受损失。

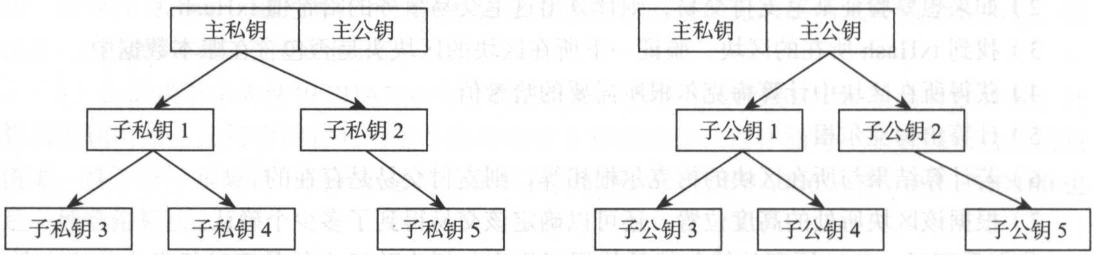
接下来我们再来介绍一种管理多个私钥的钱包技术，即分层确定性钱包（Hierarchical Deterministic Wallets，有时也简称为 HD Wallets），这个在比特币开发的 BIP32[⊖]中有专门的建议论述。简单地说，分层确定性钱包具有如下的特点。

1）用一个随机数来生成根私钥，这与任何一个比特币钱包生成私钥没有区别；

2）用一个确定的、不可逆的算法，基于根私钥生成任意数量的子私钥。

比如比特币中使用的 SHA256 就是一个确定不可逆的算法，可以很容易使用 SHA256 设计出一个 HD 模型：SHA256(seed+n)，这个就算是类型 1 确定性钱包了。实际上，分层确定性钱包是确定性钱包的一种，目前分层确定性钱包有 Type1、Type2，还有 BIP32 规范几种类型，这些都是为了实现同一目的而制定的不同实现方法，基本原理都是类似的。

所谓的分层，除了私钥由主私钥来生成逐层的私钥以外，公钥也一样，通过主公钥生成所有的子公钥。实际上，生成的密钥本身，都可以作为根来继续生成子密钥，这就是所谓的分层了。注意，这里通过公钥生成子公钥，不需要私钥的参与，无论是主私钥还是子私钥都不需要参与。我们来看下示意图，如下：



这个特性是非常有用的，在一定程度上，隔离了私钥和公钥，可以带来不少的便捷性，具体如下。

1）备份只需要备份主私钥就行了，新增地址无须再次备份私钥。

2）可以保证主私钥的冷存储，无论增加多少个地址，只需要主公钥就可以了。

3）方便审计，只需要提供主公钥或者某个分支的子公钥，就可以查看下级的数据而又保证不能被交易。

4）有了这棵树，还可以配合权限，设定不同层级的权限，能查看余额还是能交易等。当然啦，便捷性往往都是要牺牲安全性的，缺点很明显，这种钱包，由于私钥之间是具有固定关系的，不那么随机了，因此只要暴露任何一个私钥，再加上主公钥做关联分析，就很有可能使整个树状密钥结构都泄露。

⊖ BIP（Bitcoin Improvement Proposal，比特币改进建议），这是一个面向比特币社区的公开意见簿，BIP32 就是第 32 号建议的意思。

1.4.5　比特币账户模型：UTXO

我们来认识一下比特币中交易事务的数据结构，首先看这个名词 UTXO（Unspent Transaction Output，“未花费事务输出”），老实说，第一次看到这个术语的时候，一时之间真是有些懵，如果说是未花费的余额还能理解，我钱包里有 1000，花了 200，还有 800 未花费，这是很符合通常的理解逻辑的，可这个未花费的“事务输出”是个什么意思，实际上，这与比特币中的交易事务结构是很有关系的。

为了让大家更容易理解，我们暂且先不来解析这个交易数据结构，让我们进入到一个仓库，我们知道仓库的主要业务就是进和出，仓库也会把日常的进出流水账记录下来，为了查询统计方便，除了流水账通常还会汇总一份库存表出来，举例如下：

日期	方向	品名	数量	流水编号
2017-8-1	入	毛笔	10	0001
2017-8-2	入	毛笔	20	0002
2017-8-3	入	墨水	15	0003
2017-8-3	出	毛笔	15	0004
2017-8-4	出	墨水	5	0005
2017-8-5	入	毛笔	10	0006

以上图为例，这是从 2017-8-1 到 2017-8-5 之间，仓库记录的出入流水账，为了统计方便，仓库还汇总保存了一份每天的库存日报表，如下：

日期	品名	库存	品名	库存
2017-8-1	毛笔	10	墨水	0
2017-8-2	毛笔	30	墨水	0
2017-8-3	毛笔	15	墨水	15
2017-8-4	毛笔	15	墨水	10
2017-8-5	毛笔	25	墨水	10

每天仓库在需要出库的时候，只要查看一下库存日报表就知道数量是否足够了，比如 2017-8-3 需要出库 15 支毛笔，此时查看库存表发现毛笔的库存量有 30 支，足够发出，于是就将库存表中的毛笔数量减掉 15，并且将出库明细记录在流水账中。然而，这里有一个问题，库存日报表是另外编制保存的，那就有可能发生数据不一致的情况，比如 2017-8-2 时毛笔的库存本来是 30 却误写为 20，这样导致后续的账务就都是错的了。因此在有些系统中，为了防止出现这样的不一致，索性不再另外保存库存表，而只是出一张视图统计（逻辑统计，并非实际去保存这样一个统计表）。

比特币中的交易事务过程与上述的库存进出是很相像的，某个钱包地址中转入了一笔比特币，然后这个地址又向其他钱包地址转出了一笔比特币，这些不断发生的入和出跟仓库的进出是异曲同工的。然而，在比特币中并没有去保存一份“库存表”，每当“出库”的

时候也并不是去“库存表”中进行扣除，而是直接消耗“入库记录”，也就是说在出库的时候就去找有没有之前的入库记录拿来扣除，比如 2017-8-3 时需要出库 15 支毛笔，此时系统就会去搜索之前的入库记录，发现有 2017-8-1 和 2017-8-2 分别有一笔数量为 10 和 20 的入库记录，为了满足 15 的发出数量，首先可以消耗掉 10 的这一笔，然后从 20 的这一笔再消耗掉 5 支，判断成功后，系统会直接产生一条数量为 10 的出库记录和数量为 5 的出库记录，按这样的方法，将每一笔入和出都对应了起来。

在比特币的交易事务结构中，“入”就是指金额转入，“出”就是指金额转出，为了让大家对这种金额转入与转出有一个更加通俗的理解，我们来看一幅示意图：

交易类型：Coinbase			交易编号：001
输入	输出		
挖矿奖励	序号	金额	地址
	1	12.5	Alice 地址

交易类型：普通转账			交易编号：002
输入	输出		
	序号	金额	地址
001 交易中的输出	1	6	Bob 地址
	2	6.5	Alice 地址

交易类型：普通转账			交易编号：003
输入	输出		
	序号	金额	地址
002 交易中的输出	1	2	Lily 地址
	2	4	Bob 地址

上图展示了比特币中的交易事务结构，在比特币的交易事务数据中，存储的就是这样的输入和输出，相当于仓库中的进出流水账，并且“输入”和“输出”彼此对应，或者更准确地说，“输入”就是指向之前的“输出”，我们解释一下图中发生的交易事务。

1）001 号交易为 Coinbase 交易，也就是挖矿交易，在这个交易中，“输入”部分没有对应的“输出”，而是由系统直接奖励发行比特币，矿工 Alice 得到了 12.5 个比特币的奖励，放在 001 号交易的“输出”部分。此时，对于 Alice 来说，拥有了这 12.5 个比特币的支配权，这 12.5 个比特币的输出可以作为下一笔交易的“输入”，顾名思义，这笔“输出”就称之为是 Alice 的未花费输出，也就是 Alice 的 UTXO 的意思。

2）002 号交易中，Alice 转账 6 比特币到 Bob 的地址，Alice 找到了自己的 UTXO（如果 Alice 不止一笔 UTXO，可以根据一定的规则去选用，比如将小金额的先花费掉）。由于只需要转账 6 比特币，可是 UTXO 中却有 12.5 个，因此需要找零 6.5 个到自己的地址中，

由此产生了 002 号中的交易输出，注意，在 002 号交易输出中的 Alice 地址是可以和 001 号中的 Alice 地址不一样的，只要都是属于 Alice 自己的钱包地址就可以。

3）003 号交易中，Bob 转账了 2 比特币到 Lily 的地址，过程与 002 号交易相同，就不再赘述了。

相信大家看到这里，已经基本理解了所谓的 UTXO 是什么意思，我们再来总结一下。

1）比特币的交易中不是通过账户的增减来实现的，而是一笔笔关联的输入 / 输出交易事务。

2）每一笔的交易都要花费“输入”，然后产生“输出”，这个产生的“输出”就是所谓的“未花费过的交易输出”，也就是 UTXO。每一笔交易事务都有一个唯一的编号，称为交易事务 ID，这是通过哈希算法计算而来的，当需要引用某一笔交易事务中的“输出”时，主要提供交易事务 ID 和所处“输出”列表中的序号就可以了。

3）由于没有账户的概念，因此当“输入”部分的金额大于所需的“输出”时，必须给自己找零，这个找零也是作为交易的一部分包含在“输出”中。

有朋友会问：这个 UTXO 的意思是明白了，可是就这么一条条的“输入”和“输出”，怎么证明哪一条 UTXO 是属于谁的呢?

在比特币中，是使用输入脚本和输出脚本程序实现的，有时候也称为“锁定脚本”和“解锁脚本”。简单地说，就是通过“锁定脚本”，利用私钥签名解锁自己的某一条 UTXO(也就是之前的“输出”)，然后使用对方的公钥锁定新的“输出”，成功后，这笔新的“输出”就成为了对方的 UTXO。同样，对方也可以使用“锁定脚本”和“解锁脚本”来实现转账。这个脚本程序其实本质上就可以看成是比特币中的数字合约，这也是为什么比特币被称为可编程数字货币的原因，它的转入 / 转出或者说输入 / 输出是通过脚本程序的组合来自动实现的，实现过程中还使用到了私钥和公钥，也就是公开密钥算法，所以比特币还称为可编程加密数字货币。

1.4.6 动手编译比特币源码

如果有人一直在跟你说有个煎饼多好吃，芝麻有多香，鸡蛋有多金黄，你肯定希望去看一看；如果有人一直在跟你说有首歌曲多动人，旋律有多美，歌词有多感人，你肯定希望去听一听……是的，我们说了那么多的概念、技术名词，界面也看过了，可是这么一个软件到底是怎么编译出来的呢？无论你是不是程序员，都可以感受一下这个过程，看看这个设计巧妙的软件是怎样通过源代码生成可执行程序的。

比特币的源码是公开的，并且维护在 GitHub 网站上：https://github.com/bitcoin/bitcoin，目前该源码由比特币基金会进行维护。版权类型是 MIT，这是一个很松散的版权协议，每一个对比特币源码感兴趣的人都可以自由地去复制、修改，以进行学习研究。

打开网页后，可以看到有详细的程序源码以及附带的文档说明，我们就从这里下载源码进行编译。在说明编译步骤之前，先介绍些概要前提吧，烹调大餐前得先看个菜谱不是。

首先，比特币的源码是使用C++语言开发的，因此想要深入研究源码的朋友们，最好要有不错的C++基础；其次，源码中使用了很多其他的开源库，比如libssl-dev、libevent-dev、libboost-all-dev等，因此编译的时候也需要先安装这些第三方的依赖；另外，比特币源码在Linux系统上进行编译最方便，很多依赖库都是先天开发在Linux平台的，当然其他系统上也可以进行编译。

好了，接下来，我们就开始这道大餐吧！

1. 准备操作系统环境

这里我们使用Ubuntu 16.04 LTS桌面版，关于Ubuntu的安装就不在这里赘述啦，物理安装或者用虚拟机加载安装都可以，装好系统后，首先使用如下命令更新一下系统的软件源：

```
sudo apt-get update
```

2. 获得源码

先来看下获得源码的命令：

```
sudo apt-get install git
mkdir ~/bitcoinsource
git clone https://github.com/bitcoin/bitcoin.git "~/bitcoinsource"
```

1）第1条命令是安装git命令工具，这个git工具是用来从GitHub上下载源码的，事实上，使用git工具不但可以下载源码，也可以在本机创建自己的版本库；

2）第2条命令是在当前用户的目录下创建一个文件夹，用以保存即将下载的比特币源码，读者朋友具体操作时，可以自行决定路径和文件夹名称；

3）第3条命令就是从GitHub上下载比特币的源码到创建的bitcoinsource目录中。这里有个问题需要注意，如果在git clone过程中终止了，当再次进行clone时会出错，一般会有这样的提示：

```
git clone:GnuTLS recv error(-9):A TLS packet with unexpected length was received
```

出错的原因是因为git clone并不支持断续下载，删除目录后重新创建一个新目录再clone就可以了。

除了上述的git clone命令方法外，实际上，我们可以在GitHub上直接下载源码压缩包，下载下来的文件名一般为bitcoin-master.zip，然后解压缩即可：

```
unzip bitcoin-master.zip
```

解压缩后，将当前工作目录cd到bitcoin-master中，至此就可以开始着手编译了。

3. 安装依赖库

工欲善其事必先利其器，比特币源码中使用了很多第三方的功能库，这些都是必需的依赖，正所谓一个好汉三个帮，一个篱笆三个桩，没有这些可以自由方便使用的库，使用

C++ 开发比特币软件就要复杂不少。

比如，以下 3 行命令主要安装 C++ 编译器和 make 工具：

```
sudo apt-get install make
sudo apt-get install gcc
sudo apt-get install g++
```

比如，以下命令主要是安装依赖库：

```
sudo apt-get install build-essential
sudo apt-get install libtool
sudo apt-get install autotools-dev
sudo apt-get install autoconf
sudo apt-get install pkg-config
sudo apt-get install libssl-dev
sudo apt-get install libevent-dev
sudo apt-get install libboost-all-dev
sudo apt-get install libminiupnpc-dev
sudo apt-get install libqt4-dev
sudo apt-get install libprotobuf-de
sudo apt-get install protobuf-compiler
sudo apt-get install libqrencode-dev
```

libevent-dev 是一个网络库，实现网络通信功能；libssl-dev 是一个密码算法库，提供了随机数生成，椭圆曲线密码算法等功能；libboost-all-dev 是一个 C++ 工具库，提供各种 C++ 调用的基础功能库，如多线程调用以及一些有用的数据结构等；libqt4-dev 是一个跨平台的 C++ 库，用于实现跨平台运行的软件界面，这些都是比特币源码中需要用到的功能依赖库。值得一提的是，这些依赖库也都是开源的，也就是说，比特币源码不但本身是自由开源的，使用的其他依赖库也是自由开源的，这样就方便了那些希望对比特币源码进行深入研究的朋友，可以对每一个实现细节细嚼慢咽，尽情去学习和研究。

这两行命令主要安装比特币需要用到的数据存储驱动，其使用的类型是 Berkeley DB，是一种开源的文件数据库。

```
sudo apt-get install libdb-dev
sudo apt-get install libdb++-dev
```

到这里为止，就万事俱备只欠东风啦，该准备的材料都准备好了。

4. 编译准备

这两个步骤是使用 make 工具进行编译的准备工作。

```
./autogen.sh
./configure
```

需要注意的是，在执行 ./configure 的时候，有可能会看到这样的提示，如下：

```
configure: error: Found Berkeley DB other than 4.8, required for portable
```

```
wallets (--with-incompatible-bdb to ignore or --disable-wallet to disable wallet
functionality)
```

看提示是 configure 命令执行时出的问题，大概的意思是发现 Berkeldy DB 的版本高于 4.8，我们在安装 Berkeley DB 的时候，命令下载安装的是最新版本，这个其实就是个警告而已，没什么影响，提示中也给出了解决方法，在 configure 的命令后面加上一个参数就可以了：

```
./configure —with-incompatible-bdb
```

执行完毕就可以了，接下来的工作就简单啦，直接 make 编译安装即可。

5. 编译安装

```
make
sudo make install
```

执行完毕后，就大功告成啦，接下来就可以运行比特币客户端程序啦。我们可以运行带界面的程序试试，经过这个步骤，在源码目录 src/qt/ 下生成了可执行程序，同时安装到了 /usr/local/bin 目录下。

6. 运行测试

输入以下命令：

```
bitcoin-qt
```

激动人心的时刻就来临啦！我们可以看到比特币的界面显示出来了，当然了，也可以去尝试运行 bitcoind 程序。至此，在 Ubuntu 操作系统上编译比特币源码就结束了。限于篇幅，在其他操作系统比如 Mac、Windows 上的编译过程就不再赘述了，读者朋友如果感兴趣，也可以参考比特币源码中 doc 文件夹下面的 build-osx.md 和 build-windows.md 的文件说明，分别是尝试在 Windows 和 MacOS 系统上的编译。

7. 使用 IDE 管理源码

按理说到这里也没什么可说的了，编译完成了，运行也可以了，不过有没有觉得哪里不太爽呢？对了，缺少一个 IDE（Integrated Development Environment，集成开发管理），这么多的文件，用文本编辑器一个个看，要看花眼了。好，接下来我们就安装一个 IDE 工具来管理这些源码，比特币系统是使用 C++ 开发的，图形界面部分使用的又是 QT 组件，那就选择 Qt Creator 吧，本身也开源，而且跨平台，对 C++ 的编译支持也非常好。由于上述的源码编译是在 Ubuntu 下进行的，因此，我们仍然在 Ubuntu 下进行安装设置，还是按照步骤来一步步说明吧。

（1）准备 Qt Creator

可以直接到 Qt Creator 官网下载，Qt 分为商业版和开源版本，我们使用开源版本即可，

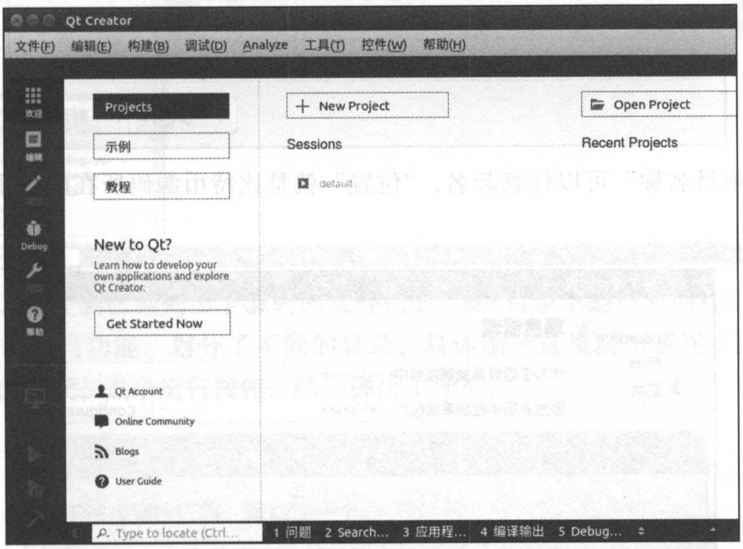
下载后得到一个文件 qt-opensource-linux-x64-5.6.2.run，读者朋友自己下载的时候，还可以选择在线安装版和离线安装版，这里下载的是离线安装版，进入到文件所在的目录，执行如下命令：

```
chmod +x qt-opensource-linux-x64-5.6.2.run
./qt-opensource-linux-x64-5.6.2.run
```

第 1 行命令是给安装文件赋上一个执行权限。

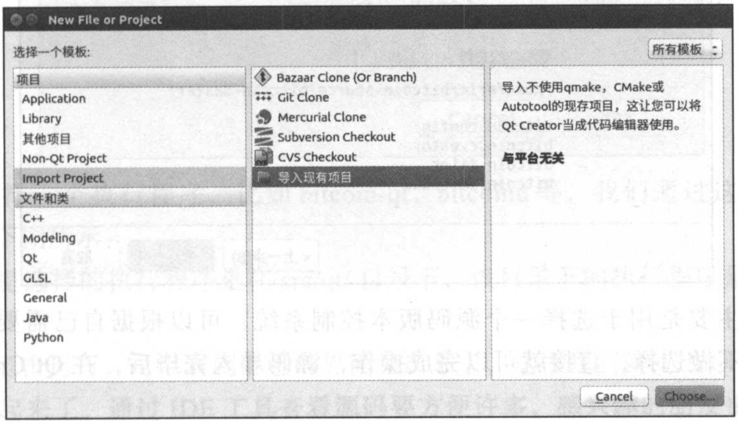
第 2 行命令是执行安装。

安装完毕后，可以打开 Qt Creator，见到如下界面：



（2）导入源码项目

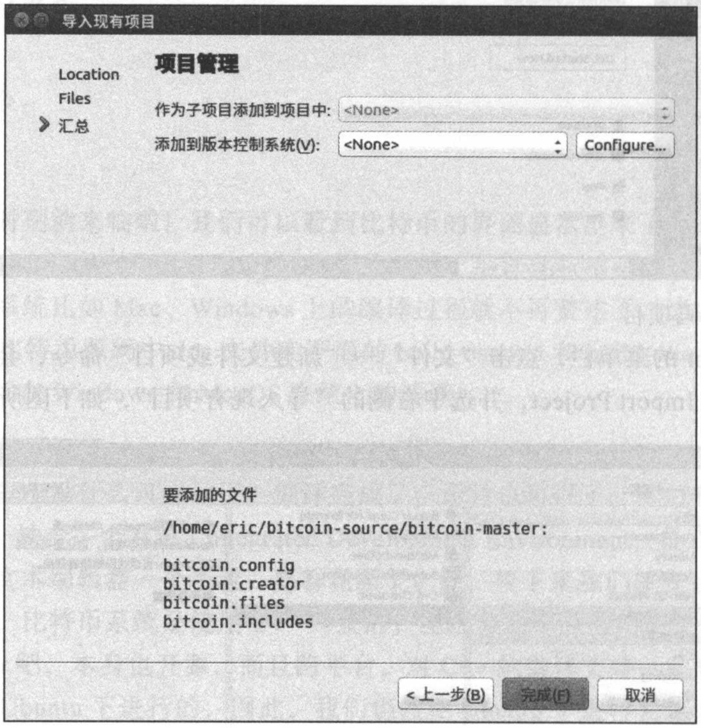
在 Qt Creator 的菜单栏，点击“文件”→“新建文件或项目”命令，会弹出一个向导窗体，选择其中的 Import Project，并选中右侧的“导入现有项目”，如下图所示：



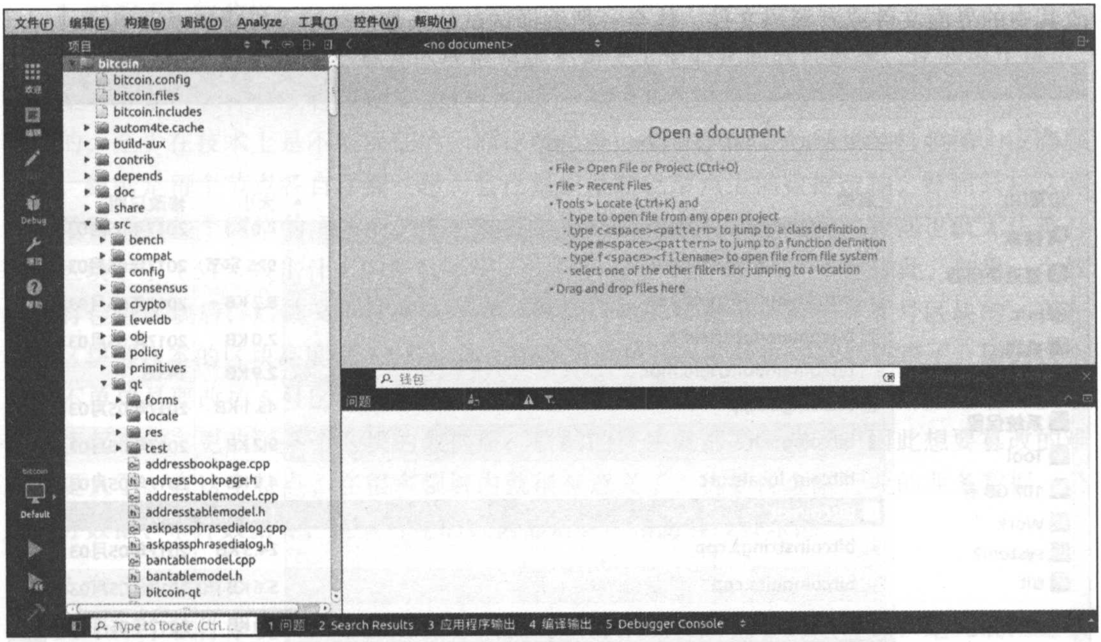
接下来就是选择我们的比特币源码所在目录，也就是需要导入的项目。



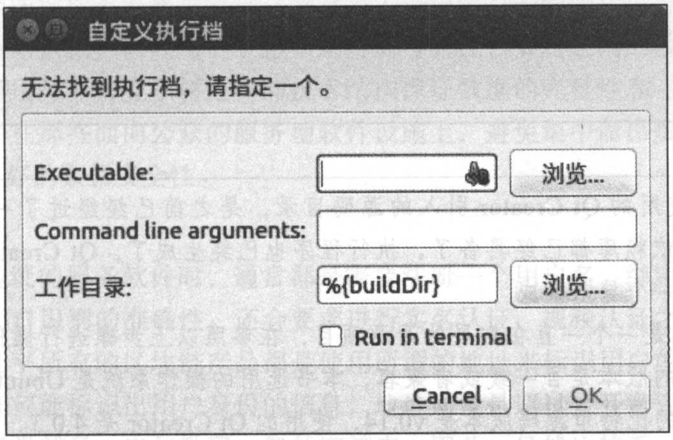
图中的“项目名称”可以任意起名，“位置”就是比特币源码所在的目录。选择完毕后继续。



这个界面主要是用于选择一个源码版本控制系统，可以根据自己需要选用，这里只是演示，因此不做选择，直接就可以完成操作，源码导入完毕后，在 Qt Creator 中的展现如下：



可以看到，在左侧已经列出了源码的文件列表，src 目录下是所有的代码文件，可以看到，根据不同的代码功能，划分了不同的目录，具体细节这里就不赘述了。到了这一步，可以运行一下试一试，点击运行按钮。咦，弹出了什么？

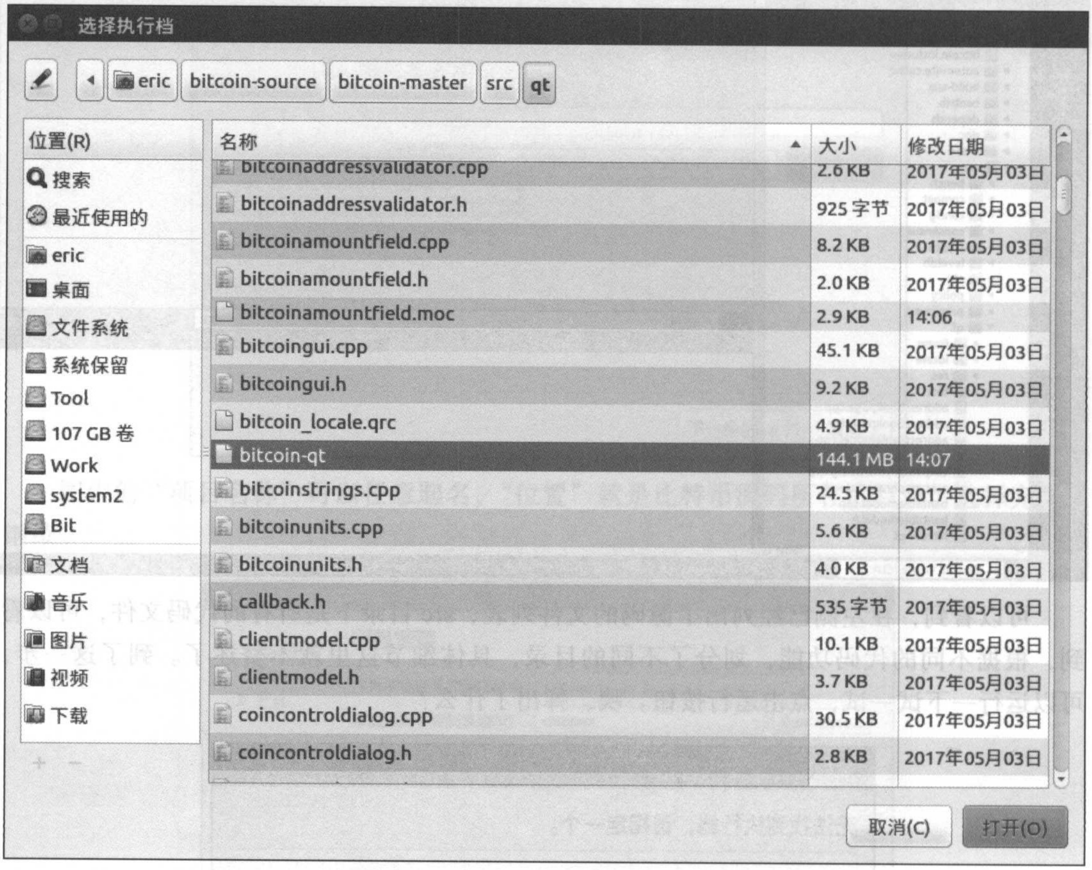


这是要选择一个执行程序，比如 bitcoin-qt、bitcoind 等，我们通过这个对话框选择 bitcoin-qt，如下图所示：

注意，这里选择的执行程序是在 src/qt/ 目录下，该目录下的执行程序是通过源码编译直接生成的。

选择后，点击运行，可以看到熟悉的界面又出来了，这样我们就使用 Qt Creator 将比特币的源码管理起来了，通过 IDE 工具查看源码要方便许多，感兴趣的朋友也可以尝试着修

改其中的界面文件或者源码文件，体会一把编译调试的乐趣。



小提示 ① 我们使用的 Qt Creator 引入的源码目录，是之前已经经过了一系列步骤编译过的，因此依赖库都已经具备了，执行程序也已经生成了，Qt Creator 就像一个外壳，只是做了一个导入集成。

② 比特币是一个一直在发展的开源项目，在参照以上步骤进行操作的时候，一定要注意选择的版本是否一致或者兼容，本书选用的操作系统是 Ubuntu 16.04 LTS 桌面版，下载的比特币源码版本是 v0.14，使用的 Qt Creator 是 4.0.3。

1.5 区块链的技术意义

要了解区块链的技术意义，我们只要来整理一下区块链系统的特点就行了，我们来一一说明一下吧。

1. 数据不可篡改性

为什么数据不可篡改呢？首先区块链系统不是一个中心化的软件设施，比如说比特币，如果是一个单机软件，或者说是被某一个人某一家机构控制的，那肯定是谈不上数据不可篡改的，至少在技术上是不能保证的，而比特币是一个 P2P 的对等网络结构软件，没有服务器，数据是每个节点各自存储一份，自己最多把自己节点上的数据改掉，然而只是这样的话是得不到整个网络的承认的，无法被其他节点验证通过，修改后的数据也就无法被打包到区块中了（51% 攻击什么的暂且不提，这是另外一个话题了）。不但如此，如果一个数据被打包进区块后，后续又连续确认了多个区块，比如数据是被打包进 5 号区块的，现在整个区块链账本的区块高度是 10 号，那么想要更改掉当时的数据的话就更难了，因为这个时候不单单是要改掉 5 号区块的数据，后续的区块都要变动，因为区块之间是通过区块哈希连接起来的，更改了某个区块的数据后，后续的区块就都要更改了，因此想要篡改的难度就更大了。有这个特点，在很多领域内就很有意义了，比如说金融行业的业务数据、公众政务数据、审计数据等，这些行业的数据都是有严格防篡改要求的。

2. 分布式存储

对于一个软件系统，一个需要保存数据的软件系统，最担心的是什么？毫无疑问是数据丢失。传统的软件设计架构，再怎么考虑数据备份或者数据库集群等，也总是不能很好地保证数据的安全，要么就是需要运营者投入大量数据备份设备或者数据库集群设施等，无论如何也不是一个廉价简约的方案。在区块链系统中，每个运行的节点都拥有一份完整的数据副本，这样的设计不但使得数据存储避免了单故障点的问题，还可以让每个节点能够独立地验证和检索数据，大大增加了整个系统的可靠性，节点之间的数据副本还可以互相保持同步，并使用类似梅克尔树这样的技术结构保证数据的完整性和一致性。这种分布式的结构很适合用在那些面向公众的服务型软件设施上，避免集中而昂贵的专用服务器配备，也具备相当良好的数据安全性。

3. 匿名性

我们在使用传统的服务软件时，通常都是需要注册一个用户名，绑定手机号、邮箱什么的，为了加强用户识别的准确性，还会要求进行实名认证、视频认证之类，然而在区块链系统中，目前几乎所有的区块链产品都是使用所谓的地址来标识用户的，仅此而已，不再需要提供其他任何能标识出用户身份的信息，地址通常是通过公开密钥算法生成的公钥转换而来的，这通常就是一串如乱码一般的字符串，因此，虽然比特币、以太坊等这些公链系统的数据是完全公开透明的，可我们却并不能知道背后的操作者是谁，不但如此，每个使用者还可以创建任意数量的地址，只要你愿意，可以每一次都使用不同的地址来进行转账等各种操作，也可以将自己的资产分散在众多的地址上，这就实现了一种用户身份的匿名性。那么，匿名性有什么用途呢？我们知道，在很多场合，如转账支付，比如收款、创建链上资产等，这些都是比较隐私的行为，匿名性在很大程度上可以满足这些隐私安全

的需求，不但是对个人，对于企业这样的商业环境，也是有同样的需求的。

4. 价值传递

这大概是区块链系统中最重要的一个特性了，所谓价值，就是泛指各种资产，比如货币资产、信用资产、版权资产以及各种实物资产（如黄金等），所有这些资产在本质上其实都是一种信用或者说信任。比如，货币，我们之所以愿意通过劳动来获取货币，是因为相信使用这些货币可以交换到需要的商品，这种信任是由政府担保的，也因为有这种信任的担保，出现了各种本身不具备太多价值但是却附带有信用价值的资产形式，比如纸币、支票、汇票等；还有版权，一首歌曲、一幅画、一本书等都有版权，版权也是一种资产，是具有价值的，这个价值从何而来，版权的背后是创作者的劳动投入，这个就是价值，国家通过法律保护这种价值，因此就能够在市场上流转传递了。在这些价值资产的转移过程中，都需要一个重要的条件，那就是信用的保证。我们需要政府、银行、担保公司等这些第三方的机构来提供信用保证，只有在这些信用保证的前提下，我们才能完成各种交易。即便是互联网发展起来后，虽然可以通过互联网方便传输各种数据（图片、视频、音乐等），但是这些数据的价值仍然需要这些第三方来保证，互联网本身并不具备价值保护的机制，而如果要在全世界范围内进行交易那就更麻烦了，涉及不同的法律，不同的价值认定规则，不同的支付方式等，那就得需要更多的机构来提供交易的保证，代价昂贵且相当麻烦。区块链能改变这种情况吗？

我们先看比特币这个例子，比特币是一种数字资产，它是由比特币软件组成的网络来维护的，在这个网络中，不需要其他的第三方，自己可以根据规则发行比特币，并且能确保发行的比特币是具有价值的（工作量证明），而这种价值的认定是通过网络中所有的节点来自动进行验证的，节点之间达成共识就算是认可了，整个过程都是自成一个体系来运行的，人们在转账交易比特币的时候，价值就发生了传递，而如果将其他的资产比如股票、合同、版权、债权等锚定比特币，那么其他的资产也就能方便地进行传递转移了。我们可以发现，区块链系统是可以自己创造信任机制的，在这样一个无需第三方的信任环境中，可以大大简化各种资产交易的过程，降低交易成本，并且由于区块链系统是一个分布式的系统，节点可以遍布全球，那就可以实现无边界的价值传递了。

5. 自动网络共识

日常生活中，我们有很多事情需要双方或者多方达成共识，比如签订一份买卖合同，买入一笔债权，担保一份交易或者购房按期还贷、众筹资金管理等，在传统的模式中，这些需求是如何提供服务的呢？比如签订合同，那就需要双方签名，必要时还需要律师审阅，公证处公证；比如担保交易，除了签名外还需要提供资产余额证明；比如购房还贷，需要有收入证明同时也需要还贷者签名；再比如众筹的资金管理，就更复杂了，需要记录每个参与者的资金项，还需要跟踪众筹资金的流向。凡此种种，这些事情在达成共识的过程中，

都需要做各种确认。

这种共识可以通过网络来自动地进行吗？如果可以，那该省多少事啊。我们还是来看比特币的例子，比特币从发行到转账交易，都是由网络中的节点自动进行身份认证和一系列的检查的，检查通过后就达成了网络共识，一笔交易就算是确定了，各个不同的节点之间达成共识的过程不再需要我们去签名，去按指纹或者去打一份什么证明了，因为每个节点都遵守一份共同的约定规则，只要一项交易符合所有的约定规则就能被确认，每个节点都确认，大家就一致认同了。那么，除了比特币这种转账交易可以自动达成共识外，其他的事务也可以吗？当然是可以的，上述提到的各种商业或者金融活动，都可以通过区块链上的智能合约来实现。区块链系统中的各个节点独立地验证智能合约，共同达成共识，如果能将这种机制应用到商业、金融、政务等领域，那将提高多少效率啊。

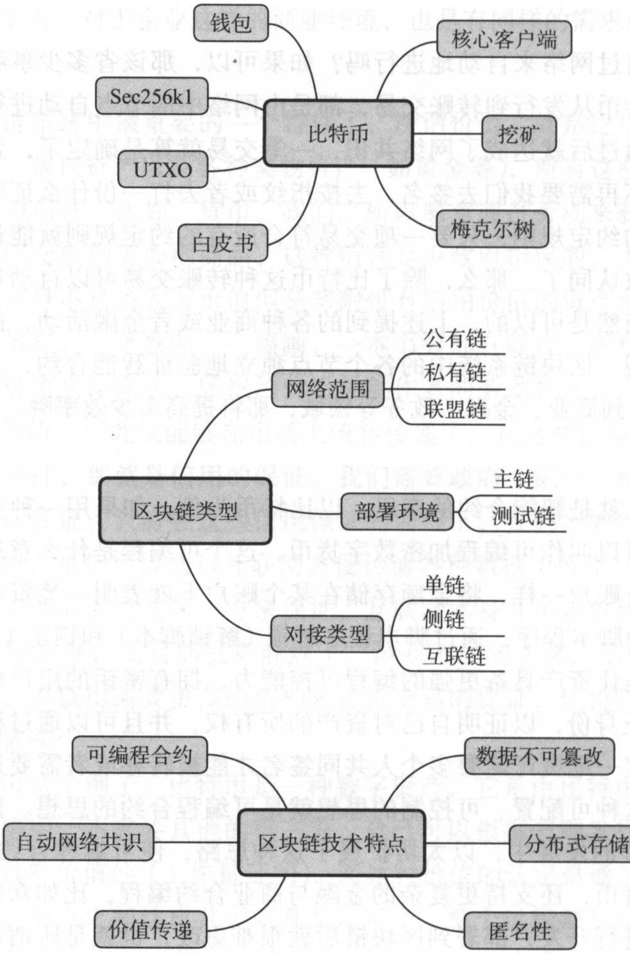
6. 可编程合约

可编程合约，也就是智能合约的意思，以比特币为例，如果用一种更加技术的称呼来描述比特币的话，可以叫作可编程加密数字货币，这个可编程是什么意思呢？在比特币系统中，并不是像银行账户一样，将金额存储在某个账户下就表明一笔资产是某个账户拥有的，而是使用了一种脚本程序，通过脚本程序解锁（解锁脚本）和锁定（锁定脚本）一笔资产，简单地说，就是让资产具备更强的编程可控能力，拥有密钥的用户可以提交自己的签名信息让脚本来验证身份，以证明自己对资产的所有权，并且可以通过程序设定对资产的管理方式，比如设定一笔资产需要多个人共同签名才能被转移或者需要达到某个条件的时候才能被使用等，这种可配置、可控制的思想就是可编程合约的思想。比特币展示了这种新颖的思路，在后续的发展中，以太坊扩展了这种思路，使可编程合约进一步发扬光大，不但支持加密数字货币，还支持更复杂的金融与商业合约编程，比如众筹、担保等。这种合约使用脚本语言进行开发，部署到区块链后就很难更改，也就是所谓的代码即法律。区块链系统具有数据的不可篡改性、价值传递能力，加上可编程合约，就能完全地支持商业环境下的各种合约需求，无论合约中有哪些条条框框，写在纸上不如写在代码中，部署在区块链上，公正透明而且能够刚性执行，更主要的是，这样的合约可以覆盖全世界，因为脚本编写的合约是不分国界的。

1.6　知识点导图

比特币是区块链技术的一种应用，而区块链的概念也是通过比特币带出来的，因此要理解区块链技术，可以从理解比特币开始，后续的各种其他区块链基本可以看作在比特币基础上的衍生扩展。比特币被设计为一种数字货币，但是对于一类技术而言，区块链技术的应用场景远不止是数字货币，我们可以结合技术特点，思考在各个领域中可能的应用。

我们来看下本章的思维导图，作为给大家的总结。


比特币
钱包
Sec256k1
UTXO
白皮书
核心客户端
挖矿
梅克尔树
区块链类型
网络范围
公有链
私有链
联盟链
部署环境
主链
测试链
对接类型
单链
侧链
互联链
区块链技术特点
可编程合约
自动网络共识
价值传递
数据不可篡改
分布式存储
匿名性

第 2 章 Chapter 2

区块链应用发展

2.1 比特币及其朋友圈：加密数字货币

抽象地说，加密数字货币其实就是一些开源的区块链技术构架及其生态工具，在完成一个个类似大富豪或虚拟城市建设一样的游戏，允许人们在这个逻辑结构复杂但完整的游戏里面根据完整的用户管理，去按照一定的逻辑生成、管理、交换、流转，甚至销毁一个个可分拆的数字单位。而这样的数字单位我们通常叫积分、代币、币或加密数字货币。

我们都知道区块链其实是一个完全分布式的，点对点互通的软件网络。在这个网络里，人们使用加密技术可以安全地发布应用、存储数据，并且可以很方便地流转价值。网络上的价值是可以跟现实中的真实货币对接的，所以区块链也是实实在在的价值网络。精密的加密技术本质就是将消息进行编码，使别人不能解密。在比特币、比特币现金、以太坊、莱特币等众多加密数字货币所构成的价值网络中，加密技术发挥核心作用，就是加密技术把价值网络中成千上万的计算机连在一起构成一个安全的计算环境，一个大型的分布式超级计算机。这个千万台小机器构成的超级计算机计算和维护着一个账本系统、一个价值网络，没有中央集权节点，没有单一的所有者，节点之间彼此几乎或者完全是对等平权的。

由于比特币最初的设计不是考虑全球性的大规模分布式应用系统的构架，在不断应用过程中，逐渐有很多性能方面的不足暴露出来，这样人们就会根据实际情况对比特币的技术进行扩展，从而产生了很多修改过的、类似比特币的加密数字货币。随着比特币及其家族逐渐发展壮大，各种利用类似比特币技术发展起来的非正统数字货币（bitcoin alternatives, altcoin），也就是俗称的“山寨币”层出不穷。主流的比特币也在 2017 年 8 月 1 日因为硬分叉，分化出一个比特币现金（BCC）。

通常很多人会说，比特币的价值背后并不像贵金属一样有使用和劳动价值支撑，比特

币其实是没有任何实际价值在支撑，这个说法其实在一定程度上（也就是理论上）是说得过去的。同样的道理也适用于现代社会的各个国家的法定货币。现代国家法币其实也是没有任何实际价值支撑的。但是唯一的差别在于法币由国家公权力来背书：法币是国家信用的背书，有些法币（比如美元）还往往是诸如原油等各种大宗特殊商品的标的物，所以大家愿意用法币去交换价值。

今天，以比特币为代表的加密数字货币，还是以法币作为价值对标，还没有能完全做到脱离法币独立存在。人们也会尝试着将数字货币推广应用成类似 VISA 卡、Master 卡那样的信用卡产品。如果未来由政府和大型机构支持，创造一个足够大的、完全基于加密数字货币的巨型市场或全球市场，各国央行也加入其中，完全用加密数字货币作为通行货币是可能的。已经有一些国家在这方面的尝试了，开始使用比特币来交换价值，购买产品和服务。

在中国和世界范围内还有各种各样打着加密数字货币旗号的各类传销币，也有部分传销币是基于真实的比特币技术的，更多的是伪数字货币而不具备加密数字货币性质的，这些我们不在这里阐述。我们下面简单介绍几个，让大家能对各种加密数字货币有一个清晰的认识。

2.1.1 以太坊

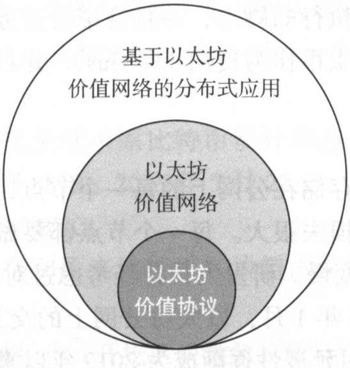
以太坊（Ethereum）[⊖]是一个开源的、基于区块链技术的、具有智能合约功能的公开分布式计算平台。以太坊有自己的编程语言，智能合约（脚本语言）是以太坊的最大亮点。以太坊提供了一个去中心化的"图灵完备"的虚拟机——以太坊虚拟机（Ethereum Virtual Machine，EVM），这个虚拟机可以将分散在全网的公共节点组合成一个"虚拟"的机器来执行这个图灵完备的脚本语言。

通常我们说以太坊（Ethereum），其实包含三层涵义：

- 以太坊价值协议；
- 由以太坊价值协议搭建起来的以太坊价值网络；
- 在以太坊价值网络上运行的分布式应用及其生态。

以太坊也发行数字货币以太币来支持技术生态。以太坊的数字货币以太币（Ether）可以用来在以太坊价值网络的节点间传递，同时也可以用作参与节点共识计算活动的"助燃剂"，俗称"汽油"（Gas）。Gas 是以太坊内部交易成本机制，用来防止过度无用交易，防止垃圾交易和网络资源浪费。每一笔交易背后都包含着成本，这点让以太坊成为众多去中心化的分布式应用喜欢的底层区块链基础构架——任何流转都有费用，可以打上价格的标签出售！

⊖ 以太坊是由一个叫 Vitalik Buterin 的俄罗斯裔加拿大数字货币程序员和研究人员在 2013 年提出来的，于 2015 年 7 月 30 号正式发布。正式的官方研发机构后来是由一个在瑞士注册的以太坊基金会完成。



（1）以太坊的版本

正式的发布版本是 Frontier 版本。之前的版本主要是概念验证版，统称为 Olympic 版本。正式的稳定版本是 Homestead，稳定版本包含了交易处理、交易费用和安全性等特征。Metropolis 版本的使命是减少以太坊虚拟机 EVM 的复杂度，让智能合约开发更简单、高效、快捷。Serenity 版本则打算将共识算法从现在的通过硬件算力来决定的工作量证明（Proof-of-Work，PoW）转到权益证明（Proof-of-Stake，PoS），并致力提高以太坊的分布式计算高可用、高可延展能力。

2016 年因为去中心化自治组织 DAO 项目资产被盗事件造成以太坊硬分叉分成现在的以太坊 ETH 和经典以太坊 ETC。

以太坊的版本信息总结如下：

版本	代号	发布日期
0	Olympic	2015 年 5 月
1	Frontier	2015 年 7 月 30 日
2	Homestead	2016 年 3 月 14 日
3	Metropolis	2017 年 9 月
4	Serenity	待定

（2）以太币

以太坊区块链中的价值代币叫以太币，在加密数字货币交易所中挂牌一般是 ETH。以太币用来支付以太坊价值网络中的交易费用和计算服务费用。

（3）以太坊虚拟机

以太坊虚拟机（EVM）是以太坊智能合约的运行环境，正式的 EVM 定义由 Gavin Wood 撰写的以太坊黄皮书做了详细的描述。EVM 建立一个沙盒，将运行环境与所寄宿机器的文件系统、网络和各种运算进程隔离开了。目前这个 EVM 已经正在被很多种编程语言实现。

（4）智能合约

智能合约是以太坊的灵魂，它承载着不信任节点之间传递价值逻辑的使命。在以太坊

价值网络中，智能合约是自动执行的脚本，并且是带着业务和资产的状态进行流转的。智能合约的发布、流转都需要以太币作为费用（即 Gas），并且能被多种图灵完备的编程语言实现。

（5）以太坊的性能

以太坊所有的智能合约都存储在公网上的每一个节点，以保证公正、透明、去中心化和不被篡改，当然也导致性能损失很大。每一个节点都要做大量的计算去流转和存储智能合约，导致全网价值流转速度变慢。研发人员曾经考虑过对区块链数据进行“分区”存储，但是没有很好的方案。到 2016 年 1 月，以太坊公网上的交易处理能力大概是每秒 25 个交易。所以网络性能、高可用及可延展性逐渐成为 2017 年以来以太坊最重要的技术话题。

（6）以太坊客户端和钱包

- Geth：Go 语言实现的以太坊客户端，也是以太坊基金会的官方客户端；
- Jaxx：网页版及手机版以太坊钱包；
- KeepKey：硬件钱包；
- Ledger Nano S：硬件钱包；
- Mist：以太坊桌面版钱包；
- Parity：用 Rust 编程语言写成的以太坊客户端。

（7）企业级以太坊

以太坊独特的智能合约技术和代币发行自动化的技术使得以太坊逐渐成为很多分布式应用的孵化器，正在逐渐成为分布式去中心化应用的首选技术。

大部分人选择以太坊有两个原因：

1）应用程序工程技术开发人员选择使用以太坊技术来创建分布式产品和各类服务；

2）非工程技术人员看重以太坊及其技术可以应用在金融、保险、银行、法务、游戏、社交、政府监管、物流、物联网、人工智能等很多领域。

对比比特币及其他区块链技术构架和生态，以太坊同时极大满足了技术人员和非技术人员的共同需要。工程技术人员使用以太坊可以快速创建、设计、发布、部署和维护分布式的去中心化应用。使用以太坊开发应用不需要了解太多的密码学知识、大型分布式系统设计构架等，工程成本和技术要求相对低于比特币类技术构架。而对于非工程技术人员，可以很轻松地直接通过以太坊的区块链浏览器和价值网络，特别是可通过工程人员另行使用脚本语言进行编程修改智能合约从而对商业逻辑进行定制化，从而达到在自己的行业快速进行现代数字货币化技术升级，进行行业颠覆等。

2.1.2 比特币现金

比特币现金（BCC/BCH）是 2017 年 8 月 1 日由于比特币扩容争端而硬分叉出来的一个比特币变种。2017 年 7 月 20 日，全球比特币矿工投票，有近 97% 的矿工选择支持比特币改进动议 BIP91，计划激活隔离见证（Segregated Witness，SegWit）功能。

比特币区块链第 478 558 号区块成为最后一个共同认可的区块，而新的 478 559 号区块成为新的比特币现金第一号区块。比特币现金钱包利用一个定时机制选择在分叉之日拒绝接受主流比特币 BTC 的区块。

比特币现金最大的特点是完全继承原比特币设计和技术构架，只是简单地将区块链区块大小从原来的 1MB 扩容到 8MB。2017 年 7 月 23 日，比特币现金 BCC 期货价格是 0.5BTC，分叉当日价值下降到 0.1BTC。

硬分叉出来后的比特币现金和原来比特币区块链的主要差别如下（截至 2017 年 8 月 8 日）：

- 硬分叉后有 243 个区块被挖出（开挖区块数量比原链少 904 块）；
- 新比特币现金区块链的运营难度只是原链的 13%；
- 原链的区块链大小增长比比特币现金链多了 920.19MB；
- 挖比特币现金的收益比挖原链的收益高 30%。

2.1.3　莱特币

莱特币实际上是从比特币的源码进化而来的，或者说得更直白点，是比特币的一个简单变种，一度有一种说法是：比特币是金，莱特币是银。其相对于比特币，主要做了如下一些变更：

1）货币发行总量增加到 8400 万，比特币只有 2100 万；

2）区块出产的理论周期提高到 2.5 分钟，比特币是 10 分钟；

3）挖矿基础算法由比特币的 SHA256 变更为 Scrypt，这种算法需要使用到大量的内存，因此一度能抵御集成电路矿机的高性能挖矿。

由于莱特币只是比特币的简单变种，因此比特币具有的问题，莱特币也基本都有，比如区块扩容的问题。不过在 2017 年 5 月份，莱特币先于比特币完成了隔离见证，之后创始人李启威便宣布了莱特币的下一步战略：智能合约与原子级跨链交换。这使得莱特币的未来充满了各种可能性，而不再一直停留在纯粹的加密数字货币这个位置上。相对于比特币来说，莱特币拥有一个明确的创始人，在很多发展事项上，创始人的领导力与关注度还是很重要的。

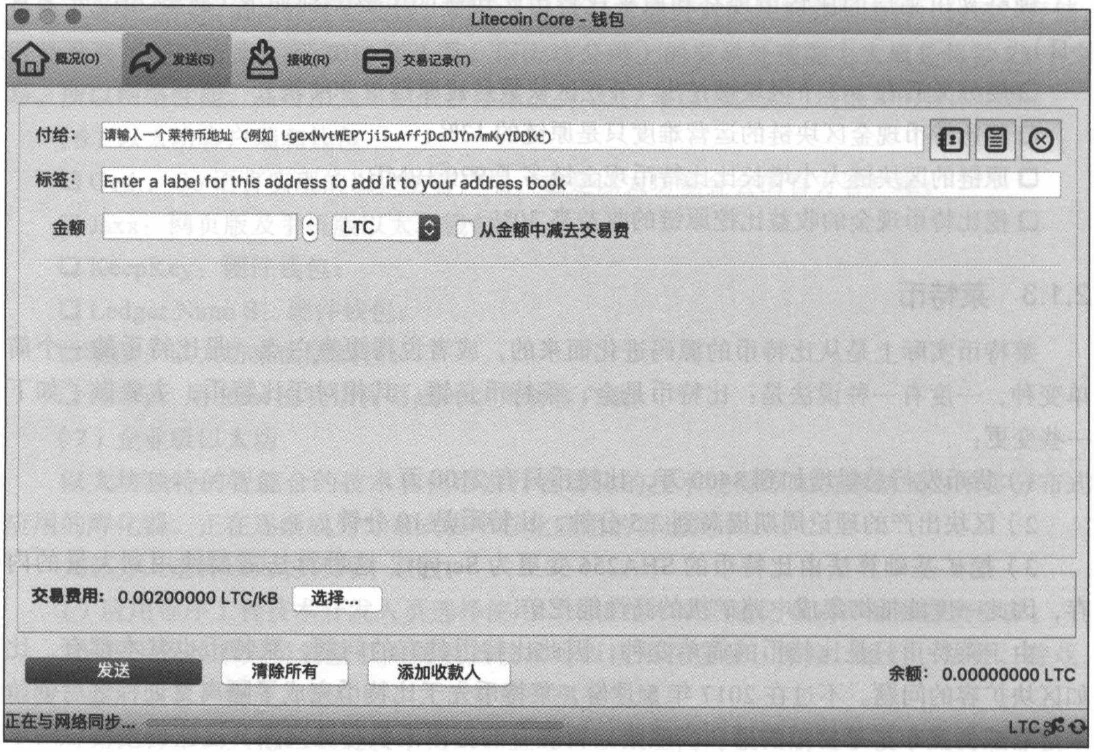
我们来看一下莱特币核心客户端，为便于大家从第一印象上与比特币做参照，我们启动了一个带界面的客户端，如下图所示。

看到界面，相信不少朋友已经发现了，这与比特币的核心客户端非常相似，莱特币本来就是源自比特币，既然都是开源系统，自然也就不去重复造轮子了。

2.1.4　零币

零币，也叫 Zcash，其前身为 Zerocoin 项目，这也是一种基于区块链的加密数字货币，并且它的总量跟比特币一样也是 2100 万枚，并且同样通过 PoW 算法进行挖矿发行。Zcash

最大的特点是：**提供了交易数据的隐匿性**。我们知道通常比特币一类的系统，其交易数据都是完全公开透明的，所有人都可以查询比特币中发生的所有的交易，而 Zcash 会隐藏交易的发送方、接收方和交易金额，只有具备查看密钥的用户才能访问到这些信息。有朋友会问，比特币难道不也是匿名安全的吗？每个人在比特币系统中只有一个钱包地址，这个地址就是一串字符而已，谁也不知道地址后面到底是谁，更何况比特币的钱包地址可以几乎无限制地创建，这还不够隐秘安全？让我们来看一看比特币的隐匿性存在哪些问题。



1）比特币地址本身具备匿名性，但是只能限制在比特币网络内部。如果要通过交易所进行法币兑换，一般要提供实名认证，比如身份证、手机号码等，这两年发生过多起比特币勒索病毒事件，实际上攻击者即使得到了比特币也难以通过合法交易所完全匿名兑换出来。

2）比特币的交易数据是完全公开透明的，虽然钱包地址本身具有隐匿性，但是所有的交易数据都是公开不加密的，通过交易的地址关联等，再加上对数据包的分析，找到对应的 IP 地址等信息，是可以有办法定位到大概的背后身份的。

目前比特币中有一些做法，比如混币可以进一步提高交易信息的隐匿性。所谓混币就是在一个交易中包含大量的输入和输出，目的就是将交易信息打散割裂，尽可能提高找出输入与输出之间关联性的难度，可以通过一些工具软件来进行高效的混币操作。当然，我们提醒大家，可以去多关注一些实现的技术原理本身，而不要轻易去实施这些做法，毕竟

这种操作会涉及一些法律上的问题。

那么，零币是依靠什么来实现的呢？具体来说，零币是使用了称之为零知识证明的机制，什么叫零知识证明呢？我们来理解一下，实际上在生活中我们也是常常会遇到的，来看一个例子。

Alice 拥有一串保险箱的密码，可是 Bob 不相信，此时 Alice 如何证明呢？如果将密码告知 Bob，让 Bob 去自己试一试，那当然可以证明，可是这样的话，密码也就泄露了。因此 Alice 决定换一个做法，她让 Bob 坐在离她比较远的地方，然后自己当着 Bob 的面打开了保险箱，以此证明自己拥有保险箱的密码，整个过程中 Bob 只是看到了一个可以证明的结果，而没有接触到密码，这就是零知识证明的基本原理了，这样的例子在生活中还有很多。

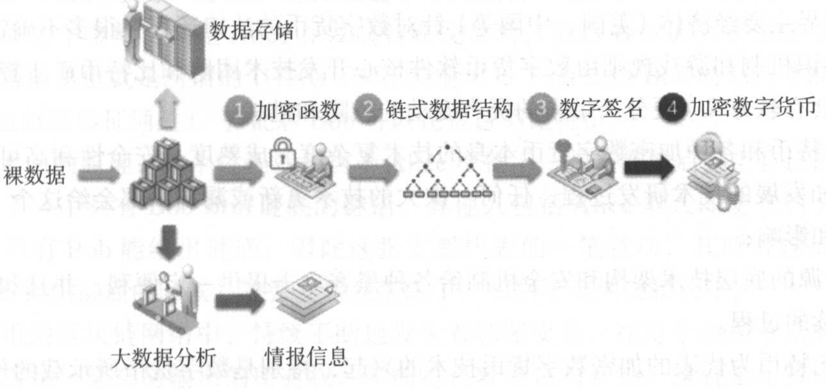
对零币系统而言，大体上是实现了一套协议，通过加解密技术，比如隐藏原地址和交易金额，生成一个字符串码，需要通过一些特有的数据才能解码获得。比特币可以将需要发送的交易转化到零币，再到比特币网络，这样就更增强了比特币交易的隐秘性。

2.1.5　数字货币发展总结

数字货币在本书特指加密数字货币。比特币、以太币还有莱特币等都是典型的加密数字货币。我们已经知道区块链技术就是分布式账本系统，账本记的就是资产的所有权，底层就是一堆数字，一堆表明资产所有权的数字。普通的代表资产背书的数字要能成为可以流通的加密数字货币，必须具备如下三个基本条件：

- 加密哈希函数（例如 SHA256）；
- 独特的、以哈希函数结果为指针的、防止篡改的链式数据结构；
- 非对称（公钥 / 私钥）密钥体系。

当普通的账本数据具有以上三个特征，普通的账本数据就具备了加密数字货币的特征。在区块链技术中，由共识机制和价值流通网络所构成的价值环境基础的烘托就可以形成加密数字货币（确切来说是可以承载价值的分布式共享账本）。



当数字的形态通过加密哈希函数、哈希链式结构和非对称密钥加密之后就会初步具备数字资产（货币）的雏形，但是这样的数字资产需要一个流转的网络，并按照一定的共识机制来保证资产的安全性和有效性，进而构筑价值网络。一般数字资产的价值网络构建共识机制包含如下5个要素（需要回答5个问题）。

（1）价值共识协议5个要素

❑ 谁来维护（存储/交换）价值交易记录账本？

❑ 谁有权决定一笔交易的合法性？

❑ 谁是初始数字资产（货币）的产生者？

❑ 谁可以修改系统（共识）规则？

❑ 数字资产交换与流转谁可以获利，获利多少，怎么获利？

当一个数字资产形态在规划好的环境中沉淀，交换并形成价值网络的时候，上面5个要素得到妥善合理的落实，加密数字货币的体系就可以构筑并运营起来了。

（2）加密数字货币正在成为"数字黄金"

参照前面提出的价值共识协议5个要素，我们来看看比特币这个"数字黄金"：

1）所有人都可以访问和维护公共账本，新交易记录向所有节点全网广播，每个节点都将接收到的交易数据写到区块链数据块中，任一节点接收新的数据块前必须对所有的交易记录进行校验（未支付，签名合法）；

2）写节点是随机的，并马上把结果数据块全网广播，接收节点将自己的哈希加到所认可的数据块中做背书；

3）初始资产分两个部分：一部分是通过创世块预留资产，大部分资产则是通过挖矿获得；

4）比特币的游戏规则由比特币核心技术团队、比特币矿工、投资者和比特币流通的商家协同整个比特币社区来决定和修改；

5）比特币的交换与流转使获得记账权的矿工、流通节点和服务提供商获得一定手续费。

然而，比特币等加密数字货币要成为"数字黄金"本身也是一个漫长的道路，其也同样面临各种挑战：

1）世界主要经济体（美国、中国等）针对数字货币的法律还存在很多不确定性；

2）共识机制和游戏规则由数字货币软件核心开发技术团队和比特币矿工控制，每次规则改变都会有软分叉/硬分叉带来的资产变动的风险和挑战；

3）比特币和各种加密数字货币本身的技术复杂度、成熟度、安全性和高可用性是一个不断进步和发展的技术研发过程，任何一次大的技术更新或漏洞，都会给这个"数字黄金"造成动荡和影响；

4）开源的底层技术架构和安全机制给各种黑客攻击提供一定便利，并且和黑客的斗争是一个持续的过程。

由于比特币为代表的加密数字货币技术的兴起，特别是数字货币所承载的价值足够大，

可能对社会经济金融环境造成足够的影响，世界上很多主要经济体国家都开始担心数字货币的影响并学习如何应对，逐步加强数字货币的研究，部分尝试将数字货币相关的活动纳入国家监管，并尝试利用数字货币技术改进传统金融法定加密数字货币。

中国央行明确将发行“数字货币”；日本内阁签署《支付服务修正法案》，正式给予包括比特币在内的数字货币合法支付地位；美国证券托管结算公司在研究和探索使用区块链技术；英国的英格兰银行在做各种基于比特币的分布式账本和数字货币的尝试。

我们正处在一个数字货币革命时代，一个新生事物的产生总是会带来各种各样的不可预见性，但是我们应该坚信，数字货币会因为其不可篡改和去中心化的天然属性及其信任机制给大家带来理念和业务形式上的变革。

2.2 区块链扩展应用：智能合约

合约管理系统实际上早就存在了，广泛地说，我们日常处理的各种商业服务都属于合约应用。比如，我们去移动充值，那就相当于与移动签了一份合约，移动确保在余额足够时系统能够自动地提高好各项服务；再如在网上商城下了个订单，支付完成后，合约就启动了，商城就开始备货、送货。某种程度上，这些合约也算是智能合约，都是通过网络技术来实现的，然而区块链系统为智能合约的实现提供了一个更加有创意而吸引人的方案。

2.2.1 比特币中包含的合约思想

在第1章中，我们了解了比特币的基本原理，这里我们专门拎出合约思想来阐述一下。作为一个分布式、去中心的网络系统，比特币在运行过程中除了发起交易外，并不需要某个人再来做一个审核确定，所有的环节比如验证数据合法性、转移所有权、打包区块等，一律都是按照既定的规则自动运行的，在这些环节里面，尤其是一个转移所有权的处理方法是很有意思的。

在比特币系统中，转账交易并不是将金额从一个账户扣减，然后另一个账户增加，而是一种更改所有权的方式。比如，Alice发送了100比特币给Bob，并不是说Bob的账户地址中存有100这个金额，而是Alice在发起转账交易时，通过Bob的公钥锁定了交易的输出，这个交易输出也就是所谓的UTXO（未花费输出），只有提供Bob的私钥才能与Bob的公钥匹配（也就是验证通过），验证后Bob可以花费这笔比特币。

抛开技术上的原理，整个过程就相当于Alice准备了一张支票然后签上自己的名字，再在支票上放了一个只有Bob知道谜底的谜语，其他人包括Alice本人即使拿到了支票也无法去兑现，只有Bob能给出谜底，因此这张支票代表的一笔款项，其所有权就转移给了Bob，当Bob提供谜底的时候，这张支票就生效了，相当于合约就执行了。

在比特币的区块链网络中，持续不断地发生着转账交易，在每个参与节点的共同见证之下，转换着每一笔交易输出的所有权，不断进行着锁定与解锁，这就是比特币系统中包

含的合约思想，多年的发展已经证明了基于区块链的这种合约设计可以用来实现价值所有权转换，由于合约设计中自带了验证机制和转换机制，加上比特币网络是面向全球的，因此比特币是一个面向全球的无边界价值传输网络或者说是价值合约执行网络。

2.2.2 以太坊中图灵完备的合约支持

以太坊是一个完全重新开发的独立的公有区块链系统，其本身也支持一种加密数字货币，称之为以太币，不过以太坊真正强大之处在于支持了用户自定义的合约编程，因此以太坊不但是一种数字加密货币，也是一个开发平台，支持全面的合约程序开发，最主要的就是支持了图灵完备的开发语言，编写的合约程序编译后是运行在以太坊虚拟机之上的，以太坊支持 4 种合约编程语言，如下：

- ❑ solidity，类似 JavaScript；
- ❑ serpent，类似 Python；
- ❑ Mutan，类似 Go；
- ❑ LLL，类似 Lisp。

官方推荐是 solidity，使用自定义合约编程可以实现各种商业逻辑，比如众筹合约、利润分配合约、担保合约、货币兑换合约等，当然也可以来实现直接的数字货币合约。在以太坊中，可以通过编写一个数字代币合约来模拟比特币，当然通过这种方式实现的数字货币是建立在以太坊的区块链基础之上的，大家可以类比操作系统之上的虚拟机。由于以太坊支持的是图灵完备的开发语言，因此几乎可以编写任意复杂逻辑的合约代码，这些被部署到以太坊上的合约程序，会受到以太坊基础区块链系统的约束，拥有公有区块链系统的一切特点，比如数据公开透明、不可篡改性等。

关于以太坊，在第 6 章有更详细的介绍。

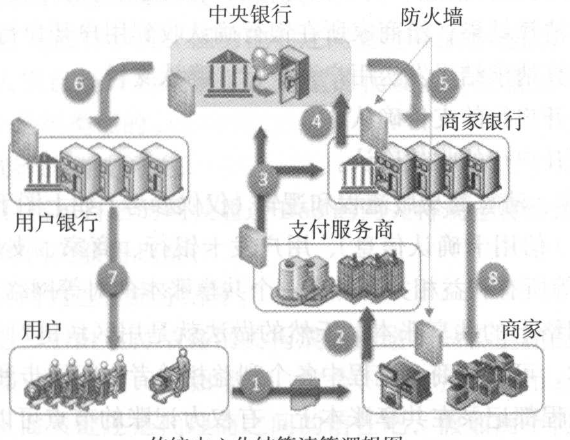
2.3 交易结算

由比特币的白皮书《比特币：一种点对点的电子现金系统》可以发现，比特币的中心功能是创造一个电子货币系统，而比特币是要建造一个交易系统并且还不仅仅是针对数字货币，而是更加广义的数字资产，换句话说，是一个基于区块链的公开透明的去中心交易系统，这是对比特币功能的进一步拓展。其目标是要创造一个数字化的自由金融体系，可以让任意种类的资产进行点对点的直接交易，不需要一个中心化的服务商。

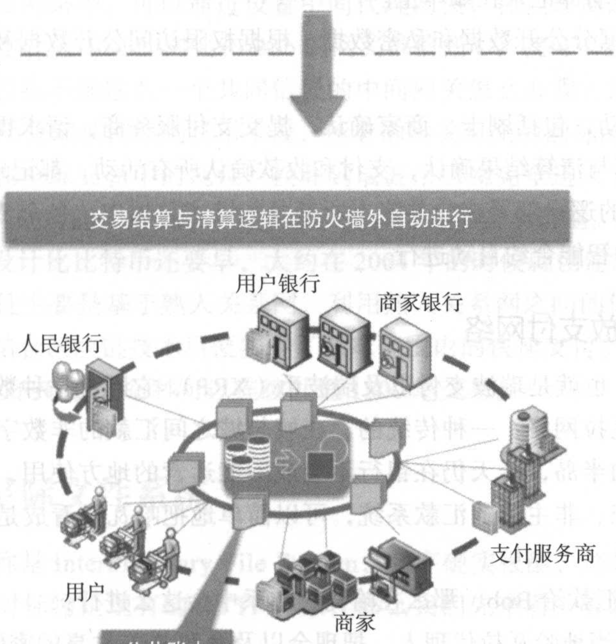
2.3.1 银行结算清算

在银行和金融业，结算与清算指一笔交易的生命周期的所有行为，包括从交易的发起，一直到交易账款结算的所有活动。所有的交易支付都必须要确保一个交易支付的承诺最终落实到真实的资产或钱款从一个账户转移到另一个账户。

传统的银行结算、清算的简单流程和逻辑（供参考），如下图上半部分所示：



传统中心化结算清算逻辑图



区块链去中心化结算清算逻辑图

传统中心化机构（银行）结算、清算流程。

1）用户到商家购买商品，刷卡，确认用信用卡或银行卡借款。

2）商家把用户支付信息（发卡机构信息）传给支付服务提供商。

3）支付服务提供商将用户银行信息发送给结算清算中心（央行或有结算清算职能的银行），同时发送给商家开户银行。

4）商家开户银行也将商家银行信息提交（央行）结算清算中心；央行根据收到结算清算请求，拿到收支双方的银行信息，确认交易合法，并做结算清算。

5）央行确认结算清算结果，给商家所在银行确认收到用户开户行的款项或信用。

6）央行将结算清算请求结果发送用户开户行，确认支付。

7）用户收到用户开户行的支付确认。

8）商家收到商家开户行的收款确认。

区块链模式的结算、清算简易版流程和逻辑（仅供参考）如上图下半部分所示。

1）用户（银行卡/信用卡确认信息）、用户发卡银行、商家、支付服务商、商家银行、（央行）结算清算中心等所有利益相关者都在一个共享账本的对等网络中。

2）建立在对等网络上的共享账本，天然的做法就是用区块链对等网络部署（私有/联盟/公有链）共享账本，可以在商业流程中各个利益相关者之间同步和传递。

3）所有的商业流程都记录在共享账本上，有权力记账的节点可以记账，需要确认的节点可以确认，所有活动都记录在账本上。

4）账本上的数据分公开数据和私密数据，根据权限访问公开数据和完全私密数据，按照应用场景分类。

5）用户交互活动，包括刷卡，商家确认，提交支付服务商，请求提交商家银行，提交结算与清算行，结算与清算结果确认，支付和收款确认所有活动，都记录在公共账本上。

6）结算与清算的逻辑部署在公共账本所栖身的链式数据中，各个节点根据共识机制与事先定义和编写好的智能合约自动进行。

2.3.2 瑞波：开放支付网络

瑞波（Ripple），也就是瑞波支付以及瑞波币（XRP），它既是一种数字货币也是一种支付协议，类似于哈瓦拉网络：一种传统的、在城与城之间汇款的非数字化方式。哈瓦拉植根于中世纪的阿拉伯半岛，今天仍在银行不会或不能运营的地方使用，它是独立于传统银行金融渠道的非正统、非主流的汇款系统，可以简单地把哈瓦拉看成是一种地下外汇来理解，我们来看个例子。

如果 Alice 想要汇款给 Bob，那么在哈瓦拉体系中会这么进行：

1）Alice 找她的当地哈瓦拉代理人，把现金以及她和 Bob 共享的密码交给代理人；

2）Alice 的代理人打电话给 Bob 的代理人，告诉他把资金发放给能提供密码的人；

3）Bob 找到他的代理人，提供密码并得到现金；

4）交易过程中的佣金可能被一方或双方代理人收取。

大家注意这个过程，Bob 通过哈瓦拉方式得到了现金，但是 Alice 和 Bob 的代理人之间并没有立即进行资金的结转，Alice 代理人仅仅告知 Bob 的代理人，Bob 的代理人（根据确认机制）转账给 Bob。我们可以看到，在这样的一个体系中，要想成功地运转这个业务，得要有如下的信任关系：

1）Alice 要相信她的代理人能够履行职责；

2）Bob 也要相信他的代理人能够履行职责；

3）代理人之间要互相履行职责，因为他们之间要进行债务还款。

看到这儿，相信大家已经开始理解了，这里的代理人其实就是做市商或者交易网关，瑞波支付的原理跟这个是很类似的。

1）Alice 登录到选择的瑞波网关；

2）将现金存入并且指示自己的网关通过 Bob 的网关将现金转给他；

3）Bob 通过自己的网关获得现金；

4）Alice 与 Bob 的网关进行债务结算。

我们看到，这就是一个由支付结算网关构成的网络，只要上述的这些信任关系不被打破，瑞波网就可以转移一切东西，比如现金、黄金、数字加密货币，甚至啤酒、大米、木材。有朋友会说，这里面关键还是代理商之间的信任吧，假如两个代理商之间不具备信任关系怎么办？在瑞波网络中，可以通过设置中间代理网关，在代理商之间建立信任链，比如创建一个中间网关 C，Alice 的网关和 Bob 的网关都信任 C，就可以通过 C 来中转了。

那么，如果两者找不到这么一个共同信任的中间网关怎么办呢？没问题，可以使用瑞波币（XRP），瑞波币是瑞波网络的货币手段，所有的网关对交易的标的物都有一个 XRP 的定价，类似于美元在全球交易中的地位。换算到瑞波币后就完全进入了瑞波网络的结算网络，可以结合瑞波网的智能合约实现不可篡改、担保等各项安全措施。

实际，瑞波的设计比比特币还要早，大约在 2004 年的时候就创建出来了，只不过一直以来都很小众，而且主要是基于熟人关系网，利用熟人关系网之间的信任实现快速的异地汇款或借贷结算，结合区块链技术后便实现了全球范围内的快速支付。目前一笔 XRP 支付交易可以在大约 4 秒内确认，每秒可以连续处理 1500 笔交易。

2.4 IPFS：星际文件系统

IPFS，英文全称是 InterPlanetary File System，名字确实很酷，这是一种点对点的分布式文件系统，它的对标物是现有的 HTTP 体系，那么我们先来看一看现有的 HTTP 体系都有哪些问题，如下。

1）中心化的服务器很容易成为性能和流量瓶颈，比如下载文件、观看视频等，当连接数多了以后，很容易速度变慢乃至服务瘫痪。这些我们都有过体会，目前任何一个使用 HTTP 访问的站点基本都是这样，最好的方式也无非就是增加更多的服务器实现负载均衡，不过这样的话，成本也就更加昂贵了。

2）站点数据不能长期保存，这是很显然的，尤其是对于长时间没什么访问量的内容，一般不是删除就是单独打包备份了，总之是不能再访问了，传统的 HTTP 服务器通常不会对站点数据做版本历史管理。

3）文件地址定位不够平滑，这个概念需要解释一下。我们知道，在通过浏览器访问站点的时候，一般都要一级域名、二级域名，然后还有文件的锚点定位，也就是说 HTTP 支持的是一个层次目录结构，这种设计一方面要求站点服务者要将自己的内容在设计开发的时候就安排好目录层次，除了实际的目录层次外，还需要设置虚拟路径等，对于一个大型站点，其目录层次是很复杂的。然而，实际上用户并不关心这个结构层次，是不是？我们访问一个站点的目的仅仅是希望快速查看内容，而不是去记住那些层次复杂的路径。

好了，我们暂且总结关于 HTTP 的这些问题点，现在来看一看 IPFS 的特点，以及通过什么样的方式能解决这些问题。

IPFS 基于内容寻址而不是 HTTP 基于域名的级联寻址，也就避免了要记住文件存储的服务器名称、路径等。IPFS 系统对每一个加入到节点的文件都计算出一个哈希值，这个哈希值可以唯一地表示某一个文件，当使用这个哈希值向 IPFS 发送文件请求时，会使用一个分布式的哈希表找到文件所在的节点，从而可以直接获得文件，这种方式最大的好处就是扁平化文件的路径层次，或者说就没有层次了，只要拿着文件的哈希值就可以，这种检索方式其实更符合人们的思维习惯。

在分布式存储结构中，文件可以切分成小的分块到不同的节点上分别存储，需要获取的时候也可以分别从不同的节点获取，这样可以大大提高文件存取的性能，实际上这也是 P2P 系统之间进行文件访问常用的一种方式。与比特币这样的区块链系统不同，文件存储节点并不需要共同维护完全一样的文件数据。

结合区块链系统的设计，拥有了数据不可篡改以及时间戳关联等特性，这样的特点非常适合应用于文件的版权保护、来源证明等场景，还可以使用基于区块链的代币来激励 IPFS 节点，这样使得 IPFS 系统带进了金融属性，使用 IPFS 的人越多，代币就会越有用武之地，而反过来就会激励更多的人去使用 IPFS。

IPFS 并不只是一个概念，实际上目前已经有很多应用了，比如 AKASHA，这是一个基于以太坊和 IPFS 的社交网络；还有 Neocities，这是一个免费的、基于 IPFS 的网页主机服务，即便 Neocities 关闭，人们也依然可以通过 IPFS 浏览到内容，这个听起来是不是有些激动。分布式文件系统与区块链技术的结合，可以产生如此多的创意应用，以至于很多人都认为，IPFS 将最终取代 HTTP 体系，我们就拭目以待吧。

2.5 公证防伪溯源

区块链技术在公证防伪方面的应用主要是利用区块链技术独特的 merkel tree 数据结构防止篡改，区块数据可溯源、非对称加密数字签名这些技术共同组成分布式点对点公证存证防伪网络及数据。我们按照图例标号来跟踪分析一个现有中心化认证公证系统的部署模式：

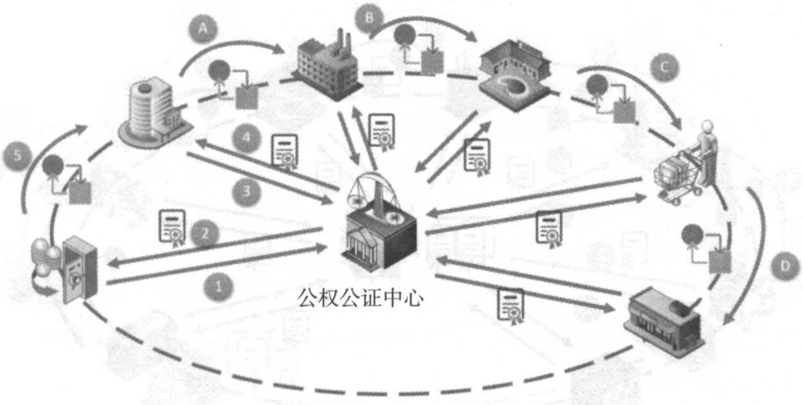
1）用户 1 向公权认证中心提交材料，申请认证证书；

2）经过审查，公证中心发放证书，用户 1 获得认证；

3）用户 1 想要跟用户 2 进行交易，用户 2 需要跟公证中心申请用户 1 提出的证明；

4）公证中心确认用户 1 的证书；

5）用户 2 要向用户 3 提出交易请求，交易中用到的来自用户 1 的证书被传递，用户 3 再次向公证中心请求确认证书，以此类推，公权公证中心是一个完全集中式的证书发放和验证机构。



中心化认证公证模式

在这个集中式的公权公证中心模式中，好处也是显而易见的：大家都相信由政府背书的专业机构，但是，这也可能存在风险，那就是：

- 单点灾难（single point failure），当这个集中式的公权公证中心出现状况的时候，全网瘫痪，大家无法获得证书服务；
- 在处理海量证书的时候，中央式的系统也会由于中央处理能力不足而造成拥堵而变得性能下降；
- 中央公证中心的安全性要求非常高，如果存在安全隐患，很容易遭到拒绝服务攻击（Denial of Service，DoS），甚至多台机器发起的分布式拒绝服务攻击（Distributed Denial of Service，DDoS）。

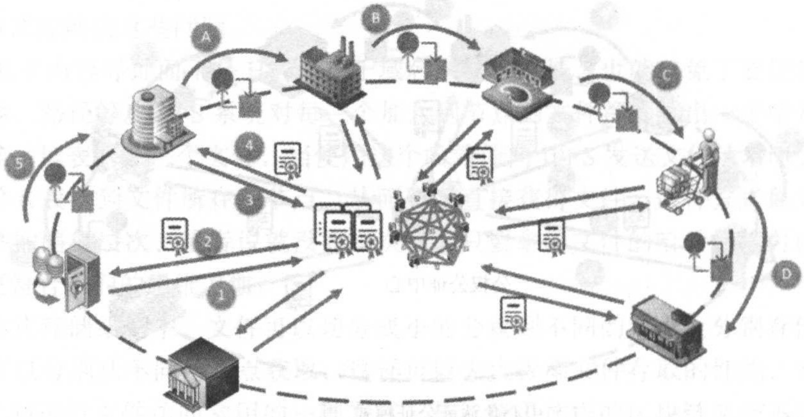
很直接的一个性能优化的解决方案就是对系统进行分布式改造，提高性能和安全性，降低被攻击而导致瘫痪的风险。区块链技术作为一种分布式计算形式，其实也可以在改造现有集中式系统的过程中逐步摸索或实施的。

现在我们来尝试着对一个集中式的公权公证中心进行分布式改造。

- 针对每一个用户，我们单独分配一个计算资源单独处理，一个单位一个处理单元。
- 在每一个处理单元的计算资源（可以是单独一台计算机）增加一个证书同步单元（模块），负责将每一个处理单元的证书进行同步，每一个处理单元都可以在同步后拿到任何单位管理的证书。
- 这样，处理单元和同步单元模式可以扩散，从一个单一的中心，根据性能需要分化

成多个中心，然后再在中心间进行证书数据同步；

- 这样原来单一中心的公权属性依然集中在认证中心，继续行使其本来应该承担的不可取代的政府或法律职能；
- 证书分发、处理、同步的功能性功能被剥离出认证中心；
- 对证书进行分布式账本技术改造，证书作为共享账本在各个单位之间产生、流转、校验。



区块链公共账本认证公证模式

注意，共享账本改造很关键，需要我们做三件事：

1）用加密函数对证书进行数字加密（采用国际认可的SHA256或中国的加密算法SM2）；

2）证书数据结构采用merkle tree来存储证书数据，保证任何改动的证书都会自动变成无效；

3）对证书进行非对称数字签名，数字签名根据证书的申请人、所有权人、发放方、校验方进行公钥/私钥的密钥管理。

经过旨在提高计算和处理性能，防止DoS/DDoS的分布式计算模式改造，再加上加密分布式账本技术改造，我们就实现了一个集中式系统的“区块链技术”改造，这样形成下面描述的基于区块链计算模型的公权公证系统。我们先说可能出现的劣势，那就是：对用来进行证书校验的密码计算要求高，如果密码被破解，全网证书（账本）就会容易被篡改；对用来进行证书同步的点对点证书通信协议的性能要求高，因为本质上分布式改造是将风险和性能同时分散到各个节点中去了。当然新系统的优势是非常明显的：

1）证书数据由于公钥/私钥非对称数字签名使得数据来源和权益清晰，证书内容及其真实性得到来自源头的保证；

2）证书内容在产生或修改的瞬间，就已经全网无差别地被同步并通知；

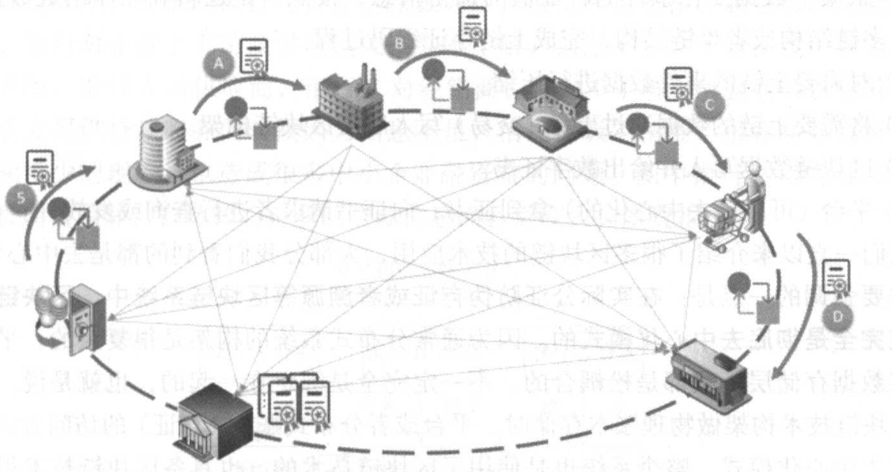
3）任何证书的改动及其改动过程都记录在案，而且可以溯源；

4）任何证书的校验都可以通过签发及确认者的公钥完成，不再需要中间方参与；

5）分布式签证校验系统导致可以完全不需要信任第三方的存在和干预，系统自动完成；

6）公证网络参与方平权，系统自治；

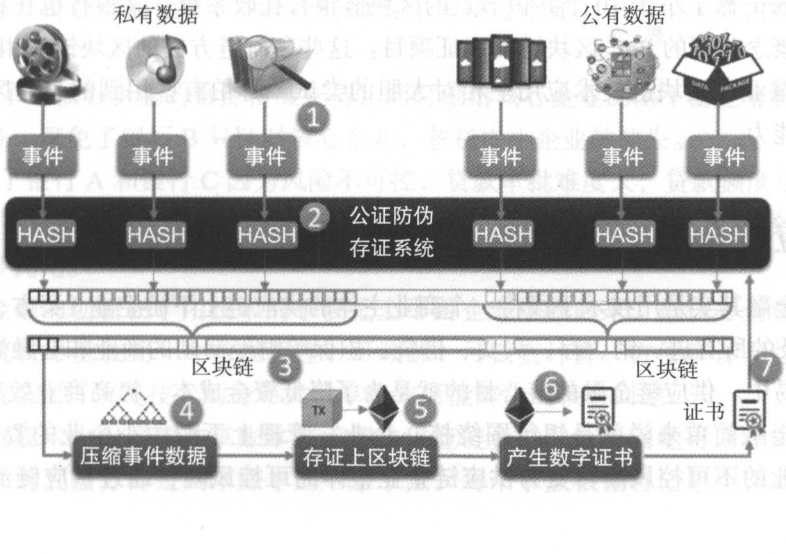
7）原公证中心（公证处）功能职能弱化，可以逐渐变成一个象征性的法律主体，更适合做其法律专职工作，法律意志体现的效率得以提高，意义更大。



区块链对等网络认证公证模式

我们再进一步看一个类似存证通的区块链公证存证防伪系统的具体实现过程，以及一些核心技术原理。这个系统的理念在于所有被认为有价值的数据，包括私有数据和公共数据，都可以存证公证并防伪。

如下图，系统从数据开始，自上而下进入系统，到达系统最底层然后从左到右完成证书签发再回到系统向上分发返回给请求用户。



1）以事件为导向，当数据发生变化时向系统发出存证、公证、防伪识别等请求。

2）外部事件的请求是分布式的，可以有多种数据源，确保系统的高可用性；当外部请求进入平台后，平台根据事件所请求的数据的变化，抓取变化特征值，将特征值的元数据进行哈希函数加密（SHA256）。到目前为止，区块链技术中只有通过哈希函数加密的或者使用类似同级加密强度的加密函数，才能保证数据的不可篡改性，否则都会留下安全隐患。

3）采集了数据变化事件的特征值的加密信息，按照一定逻辑和循序组成数据块上链，可以是多链结构或者单链结构，完成上链存证编码过程。

4）对需要上链的事件数据进行压缩。

5）将需要上链的数据通过事务（交易）写入底层区块链构架。

6）区块链数据写入并给出数字证书。

7）平台（可以是去中心化的）拿到证书，向证书请求者进行查询或发放。

我们一直以来介绍了很多区块链的技术应用，大部分我们看到的都是去中心化模式的。但是需要强调的一点是：**在实际公证防伪存证或者溯源等区块链系统中，区块链的应用并不一定完全是彻底去中心化模式的。**因为通常分布式系统的构架是很复杂的，平台访问层和底层数据存储层一般都是松耦合的。不一定完全是绑定在一起的，也就是说，在存储层使用区块链技术构架做物理账本存储时，平台或者分布式账本（存证）的访问方式可以是集中式或去中心化模式，整个系统也是使用了区块链技术的，也具备区块链技术很多优良特性。这样的区块链系统我们常常称为“存储层区块链应用”。

当然，也有去中心化理想主义者认为这种类型的区块链应用只是非常简单的几个节点，属于私有链模式，在内部网络内实验用，而且只用到了区块链最少的一个应用——分布式数据物理存储，可溯源、不可篡改的属性，也戏称这样的区块链应用为“伪区块链”应用。因为这样的区块链没有在整体上多节点来构建基于区块链的价值网络。在区块链技术应用的探索中，我们听到2017年某银行曾经尝试过一个共识节点的区块链银行核心业务系统，某银行用Fabric做了小于10个共识节点的区块链银行托收系统，某银行也在私有链上尝试了少量节点概念验证的PoC区块链信用证项目。这些私有链方面的区块链应用项目的尝试，也是中国金融业在区块链技术应用上相对大胆的尝试。哪怕有些用到的只是区块链技术公证防伪溯源能力。

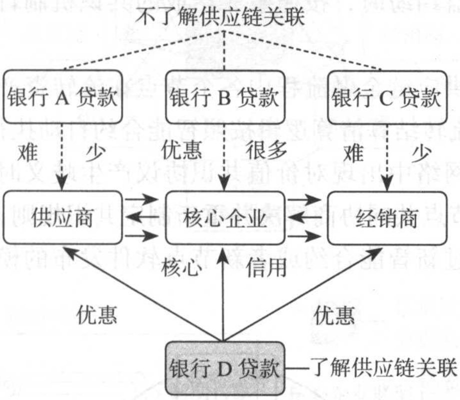
2.6 供应链金融

供应链金融其实是用技术手段将一个商业主体的供应链上下游企业（买方、卖方、金融机构）所涉及的所有进、销、存、买卖、借贷、担保等财经活动的商业和金融流程集成到一个完整的交易中。供应链金融的核心目的就是为了降低资金成本，提高商业效率。

供应链金融简单来说就是银行围绕核心企业，管理上下游中小企业的资金流和物流，并把单个企业的不可控风险转变为供应链企业整体的可控风险，通过供应链流程获取各级

核心企业、供应商、经销商的各类信息，同时利用核心企业的信用以及订单作为背书，将风险控制在最低的金融服务。同时由于充分利用了技术手段，通过信用流转而非现金流转，大大降低了现金的使用率，甚至在一定程度上消除了现金的使用；同时因为数字或区块链化的资产可以将应收和应付的账期大拆小，短拆长，甚至任意分拆，从而大大增加了信用令牌（credit token）的流转率，增加了资金的流动性进而增加了资金的利用率。

大部分情况下，传统的银行与企业金融服务中，银行和企业在产业生态供应链环节中都是单点，银行对企业上下游及供应链关系并不了解，信息不多，了解不多，甚至完全不了解。如下图，银行A对供应商，银行C对经销商，因为不了解对应企业的供应链资产细节，在给企业提供贷款服务的时候因为信息不足，信用评级不精准，很容易导致贷款审批难，贷款额度少的现象，也造成很多中小企业融资难的问题。图中银行B因为面对的是核心企业，长期信用好，自然核心企业的贷款优惠而且额度大，融资很容易。



上图银行D因为掌握核心企业的供应链上下游企业、供应商和经销商的资金与物业流动信息，对供应商和经销商的贷款可以做到精准、高效、低成本，同时也会给核心企业最好的金融服务支持和信用额度。这样银行D在了解核心企业上下游供应链信息的基础上获得的优势如下。

1）保持了对核心企业的重点精准支持，同时也给核心企业的上下游企业提供优惠、高效的金融服务，避免了银行B只针对核心企业，忽视中小企业的缺失。

2）避免了银行A和银行C因为风险不可控，贷款审批难度大，贷款额度小的误区。

3）保持了对供应链生态上下游全产业流动资金、物流信息的了解，金融效率和成本还可以更进一步优化。

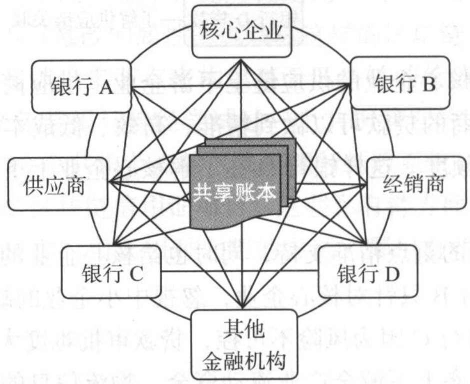
供应链金融的创新在于用技术的手段把上下游企业串联起来，把核心企业及其附属企业的应收和应付尽量放到一个首尾相连、可溯源、可核查计算、方便审计与监管的交易记录中去，这就是用区块链技术的分布式共享账本所能解决的核心问题。

用区块链技术中内置的共享账本技术解决供应链金融的核心问题可以让供应链金融全流程中所涉及的利益方都是共享账本的一个节点，每个节点根据自己原生态或被赋予的商

业属性在一个区块链网络中扮演自己应该扮演的权利和角色。所有的人都可以在经过授权或者许可的基础上读/写一个供应链金融环节中的交易数据。共享账本所承载的供应链金融信息和价值可以自由分叉与合并地进行流转和传递。

如下图，我们按照区块链价值网络的构成要素和步骤搭建供应链金融价值网络：

- 供应链金融全流程所有的利益单位和企业构成一个平权的价值网络；
- 企业按照商业逻辑达成一套供应链金融价值共识机制；
- 全网共享供应链金融共享账本，节点按照功能进行分类，规范交易与交易数据读写权限；
- 供应链金融价值网络通过 Asset Back Cyrptocurrency（ABC）的模式进行价值上链，实现资产区块链化；
- 供应链金融价值的生命周期管理（产生、流转、分拆、提现等）由智能合约制定；
- 当价值网络出现利益纠纷时，按照事先达成的共识机制自动处理纠纷，由智能合约自动执行；
- 价值共识协议规定供应链金融流程中各个节点在流转资产和记账的过程中是否有获利节点，而且最后流转结算清算逻辑按照智能合约自动执行完成；
- 当供应链金融价值网络中出现对价值共识协议产生歧义时，由平台管理方会同各个供应链金融环节的节点共同协商解决并重新制定共识规则；
- 新共识规则可以通过新智能合约或者新节点软件发布的模式进行更新，防止价值网络分叉。



在实际信用（数字资产）流转的过程中，我们可以看看下图所展示的一个流程例子：

1）供应商根据供货清单以及核心企业的票据凭证提出应收账款上链；

2）核心企业根据供应商提供凭证确认信用真实性，供应商的应收账款按照账期形成上链数字资产（可以通过挖矿或机构授信额度预充值）以上链；

3）供应商获得带账期的、以核心企业票据为背书的数字货币信用资产所有权；

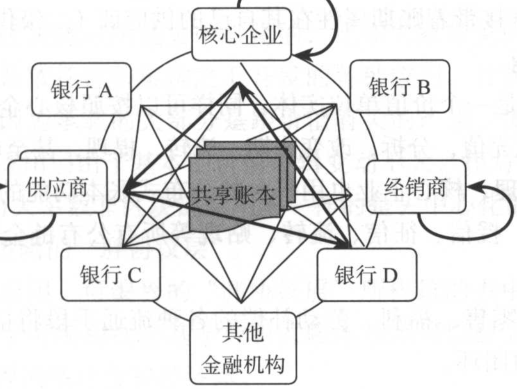
4）供应商可以根据账期到金融机构（银行或供应链金融运营平台）按照一定逻辑的提

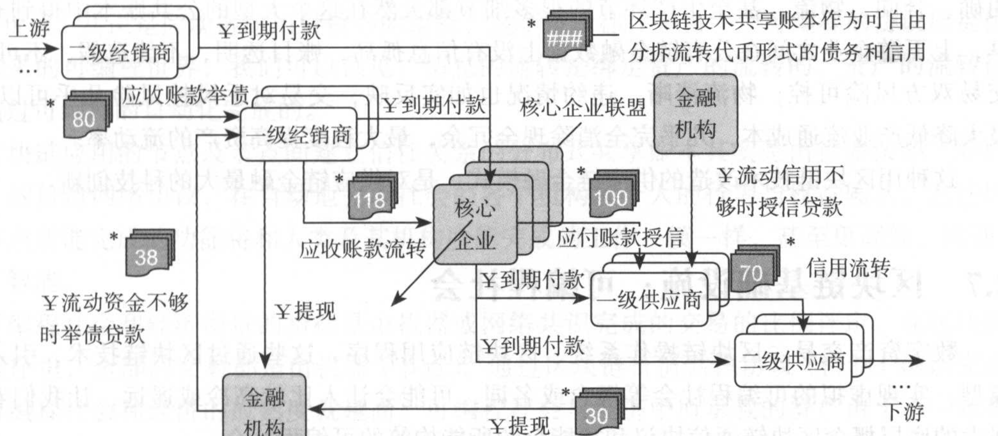
现率提现；

5）供应商也可根据数字资产信用，任意拆分去流转给自己的供应商，可以大拆小，短变长，或者直接提现，如果平台有资金或别的数字货币理财产品也可以转入数字货币理财；

6）供应链上的数字资产分割、流转、交易、提现、利息、支付都是通过内置在区块链基础架构里或者编译成可自动执行代码的清算与结算逻辑来自主完成。

所有以上步骤都可以通过区块链分布式账本技术，可以但并不必须通过智能合约来自动执行与监管。通过技术手段可以实现供应链金融交易的完美金融自治。





基于区块链技术的代币流转和分拆的
区块链供应链金融逻辑图

在上图所描述的公有链金融区块链应用场景中，我们用区块链来打造这样一个区块链公有链金融平台（仅供参考）。

1）平台和金融机构根据产业公有链中核心企业的信用，给核心企业发行（或发放）信用令牌额度，按照数字货币（令牌）发行的机制，商业票据质押的模式将核心企业价值置换

到公有链金融产业链中。

2）所有企业流转的信用令牌加上提现账期时间属性流转，信用令牌按照加密数字货币技术可信用充值，大小分拆，账期长短调整，信用令牌可流转，可提现，可作为产业流通和结算价值单位。

3）当核心企业欠其供应商货款的时候，在一定账期内的应付款，可以使用核心企业的信用令牌置换信用，并对标供应商货物价值及费用。

4）一级供应商拿到核心企业的信用令牌：第一，可以等待账期到期到平台全款提现；第二，可以在账款到期前跟平台或平台具有保理功能的企业按一定的提现率提现；第三，一级供应商可以将信用直接带着账期属性在其自己的供应商（二级供应商）确认的条件下流转应付账款给下游供应商。

5）二级供应商若是一个价值单位实体，同样可以按照核心企业公有链环境下的价值信任体系对信用令牌实施充值，分拆，改变账期，流转，提现，甚至申请额外信用的所有活动。

6）按照同样的道理，核心企业也用平台的分布式账本功能的信用令牌在上游经销商那里盘活经销商的应付、授信、征信、流转、贴现等所有公有链金融功能，全产业链形成价值闭环。

7）企业可以通过零售、福利、劳动补偿的各种流通手段将信用价值置换到个人手上，形成企业到个人的价值闭环。

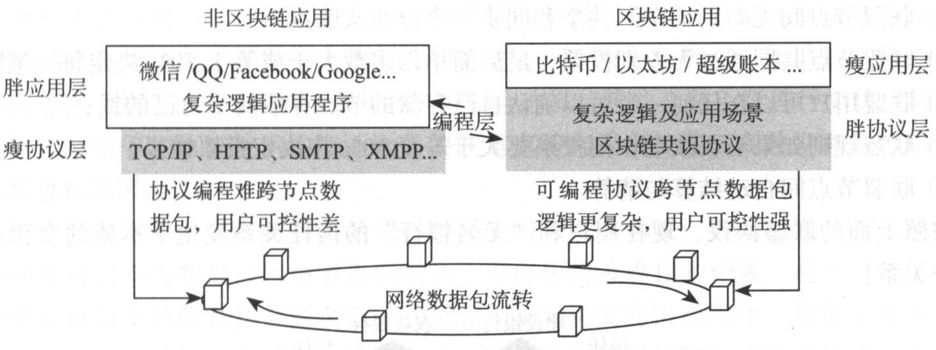
这样构筑起来的区块链公有链金融平台在整个公有链金融环节，由于上下游供应关系明确，合同，物流，甚至生产环节的很多细节都天然在这个大型的公共账本中得到有效共享，上下游企业之间在公有链金融数据上没有信息孤岛，账目透明，不能造假，不可篡改，交易双方风险可控，物流清晰，违约情况也如实反映，交易对手风控风险几乎可以消除，极大降低产业流通成本，几乎完全消除现金冗余，最大程度提高资产的流动率。

这种用区块链技术改造的供应链金融模式，是对供应链金融最大的科技创新。

2.7 区块链基础设施：可编程社会

数字资产交易、区块链操作系统、区块链应用程序，这些通过区块链技术，引入经济模型，实现虚拟的可编程社会等概念或名词，可能会让人比较高冷或遥远，让我们看看最强大的底层概念区块链通信协议和智能合约所能构筑的可编程社会。

我们先来看区块链应用及其基础构架。一般而言，我们传统非区块链应用（如微信、QQ、微博、脸书（Facebook）、谷歌（Google）等）都是将复杂的逻辑放在应用层，也就是我们看到的“胖应用层”。而在传统软件的底层则是通信协议层，这层一般指互联网通信协议（TCP/IP、HTTP、SMTP、XMPP 等），在传统的非区块链应用中，网络协议或应用协议层一般一旦确定下来很少修改，所以很多应用逻辑都是在应用层完成，而网络协议层的改动、修改或编程则很少，所以我们往往将这样的应用的协议层称为“瘦协议层”。



大部分基于互联网的区块链应用是去中心化的，比如大家常见的比特币、以太坊、超级账本等区块链基础架构及依托这些架构之上开发的各种应用。目前大部分去中心化应用在应用层逻辑相对简单，因为承载的大部分是跟价值有关的产生、流转、分拆、提现、买卖等应用。相对地，这个应用层由于基本的价值逻辑变动不大，相对需要编程的部分不多，可以说是“瘦应用层”。与大多数非区块链应用不一样的是去中心化的区块链应用的协议层往往是一个包含很多复杂逻辑的“胖协议层”。

不同于非区块链技术应用，可编程的“胖协议层”所构筑的去中心化区块链应用正在构建一个可编程社会。这个可编程社会基于分布式账本技术可以建立一个不需要第三方信任机制、彼此信任的可编程网络社会和经济体。

区块链应用有一个非常独特的特性就是其价值网络（共识）协议，如何在网络节点之间形成（价值）共识是区块链应用最核心的逻辑，这个逻辑往往是由一个可编程的协议层提供的。未来的可编程世界，我们可以预见：信息的流转是绑定资产的流转的，资产的流转往往是通过可编程的自动化完成的。

区块链应用的节点及节点间建立信任关系的分布式共享账本其实是由很多按照“胖协议层”的价值网络协议，在自动地完成社会的各个机构和个人的行为与权益确认。这些区块链节点所能完成的功能将和人类及其机构所能完成的工作一模一样，甚至更高效、准确、公平、智能。

可编程社会和经济衡量的指标是由机器或网络共识完成的交易的比例评定。在区块链应用程序里，全部的交易都是由机器（节点），通过区块链价值共识协议（机制）确认完成的。可编程社会和经济的成熟度就越高。可编程社会与经济中所承载的资产由于不一定都是按照法币来衡量的，所以也往往叫作“影子资产”。

下面举一个例子：

平时大家都相信国有商业银行，并愿意把钱存到大银行。因为信誉好、有国家背书等，但是我们也许不相信一家“无名银行”，很难有人知道无名银行的可信程度，人们对“无名银行”往往“不信任”，拒绝到“无名银行”存钱。

现在让我们假设用区块链技术将包括“无名银行”在内的银行都连接起来组成一个银行联盟，而组成联盟的联盟协议是：

1）联盟节点间无差别地全网共享和同步一个分布式账本；

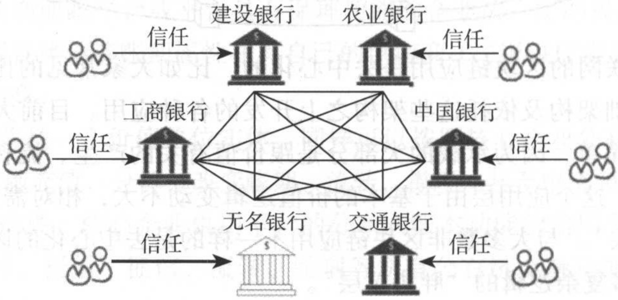
2）联盟节点共同通过无差别投票，最后简单以多数大于或等于 50% 决定每一笔账；

3）联盟用户可以在任何一个可以确认自己存款的节点存取属于自己的钱；

4）联盟规则如果需要改动，则投票要大于等于 50% 来决定投票结果；

5）联盟节点间自动结算与清算。

按照上面的联盟协议，现在用户和“无名银行”的信任关系发生了本质的变化：建立了信任关系！



哪怕“无名银行”消失了，用户的钱还是可以在任何一家联盟银行那里得到确认并存取，这就是信任机制建立的关键。而这个关键的背后，就是区块链技术建立起来的信任机制和价值传递。

分布式自治组织 DAO（Decentralized Autonomous Organization）也是可编程社会的一个很有意义的尝试。

2.8 链内资产与链外资产

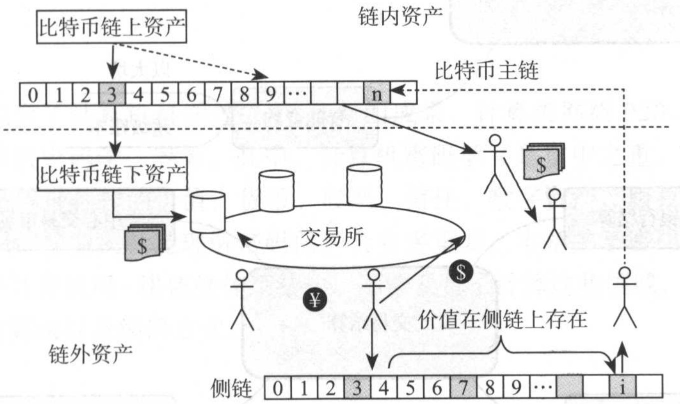
比特币的交易费用现在逐渐提高，特别是在对交易实时性有很高要求的时候，往往出现完成一笔交易所需要的费用比交易本身承载的价值还要昂贵的情况。其实有一个办法避免交易费用，就是将交易移到链外做。这就是所谓的比特币（区块链）链外交易和链外资产。要完成一笔链外交易对一个普通的用户而言相对专业企业（比如交易所）要困难一些。我们会很好奇，一笔链外资产的交易是怎么完成的呢？让我们来仔细看看链外资产的概念及其交易逻辑。

其实链外资产的概念我们光从字面上就能轻易理解，就是不在比特币主链上的资产。涉及资产的产生、分拆、流转、交易、储存等都是在区块链主链链外进行。平时我们看到的比特币等区块链交易，一般都是在链内不同的用户（地址）间进行和完成交割的。但是也不尽然，人类的智慧是无穷的，往往可以做到在链外的交易比链内的交易更频繁。**这是因为资产本身是可以对标和分拆的。有很多方式可以让交易双方在不上链的情况下完成交易。**

链外交易其实本身有不少好处。首先，可以避免价值网络上的链上交易费用，大部分的区块链应用在已经形成的价值网络上进行交易是需要交易费用（Gas）的。有些比特币交

易手续费用甚至达到 10%，特别是交易转账繁忙的时候，有时候即使加大交易费用，由于网络堵塞还是不能成交，甚至有人愿意支付超过 100% 的交易费用。如果是这样，比特币支付的优势就不复存在了，用户（或机构）可能会选择传统的金融手段进行交易。其实，这也是导致比特币等区块链技术构架逐渐将关注重点转向提高性能、高并发和高可用性问题上来的最直接原因。

链外交易的另外一个好处其实就是有时候交易可以非常快。换句话说，就是某些形式的链外交易可以不需要等待网络节点的确认，可以更快地确权和记账。此外，链下交易还可以提供更好的交易隐私甚至匿名性，甚至完全不知道交易已经发生。就像交易双方私下交易，用线下资产质押、分拆、转移等方式，最后交割汇总结算和清算而这完全在链上得不到体现。这其实也是对比特币等区块链交易的提醒，区块链技术的应用也不能完全杜绝黑箱操作，甚至有时候很多人和机构打着区块链互信公开、存证不可篡改等功能的旗号，私下做隐藏交易，内幕操作的坏事。



介绍完链外资产和链外交易的概念，我们下面来看看链外交易是如何完成的。如果交易涉及多个彼此不信任的参与方，一般可以用引入侧链的方式完成，就是将比特币主链上的价值转移到侧链上完成，当然我们要考虑符合交易双方的特殊需求。

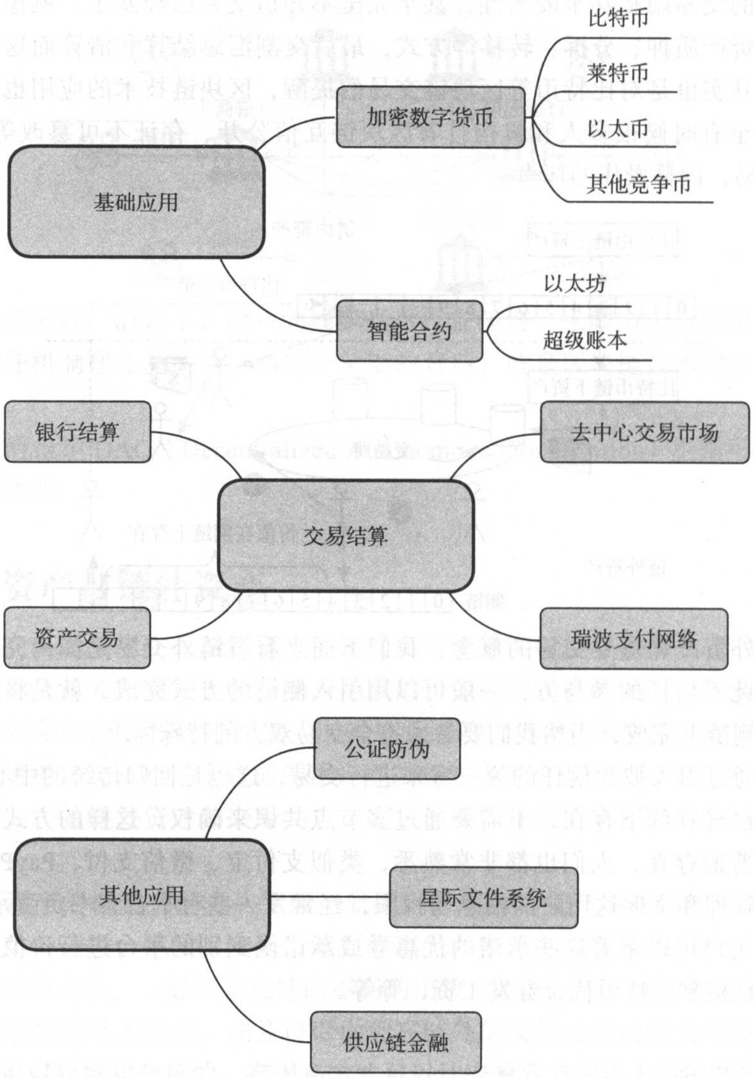
还有一种通过引入彼此信任的第三方来进行交易，这也是回归传统的中心化交易模式。因为信任机制已经在线下存在，不需要通过多节点共识来确权。这样的方式在传统金融的支付行业已经普遍存在，人们也都非常熟悉，类似支付宝、微信支付、PayPal 等。在一些交易所，因为管理和掌握这用户的信息与权限，经常发一些有平台背书负责承兑的优惠券、积分等，这样用户可以拿着这些承诺的优惠券或承诺函到别的平台进行价值转换。比如有些公司，甚至直接拿比特币代金券发工资，等等。

2.9　知识点导图

区块链技术的发展就是人们认知的发展，早期大都只是模仿比特币，通过修改比特币

的源码“创造”出另外一种币，大家的认知还停留在“币”这个概念上。渐渐地，人们通过深入理解比特币的技术原理后，发现这种技术思想只是用来实现一个“币”实在是有些局限，于是创造出了功能更强大的支持智能合约的系统，这实际上是扩展了比特币中交易脚本的概念，进而人们开始结合其他领域，发现更广阔的应用场景，比如分布式存储、去中心交易、无边界支付网络等。一类技术的产生，往往就是一类思想的革新，可以预见，区块链的应用发展会有更广阔的未来。

好，我们看下本章的思维导图总结。



在本章中，由于限于篇幅，我们不能一一介绍所有的应用场景。作为一名技术人员，真切地期望一种好的技术思想能够对社会的生产发展以及人们的生活带来促进作用。

第 3 章 Chapter 3

区块链骨骼：密码算法

区块链系统包含了计算机科学过去 20 多年的成果：计算机网络 P2P、算法、数据库、分布式系统、计算机密码学，等等。其中，计算机密码学又是重中之重。只要想到以后在区块链上跑的会是各种各样的价值：货币、股票、信任、数字资产、版权、交友信息……不需要太多的知识，普通人就能理解密码保护会有多重要。本章关于密码学的讨论、证明和陈述，完全基于计算机冯 · 诺依曼体系结构，不涉及量子计算这些领域，主要讨论区块链里用到的典型加密算法以及编码方式。

3.1 哈希算法

哈希算法在区块链系统中的应用很广泛：比特币使用哈希算法通过公钥计算出了钱包地址、区块头以及交易事务的哈希值，梅克尔树结构本身就是一棵哈希树，就连挖矿算法都是使用的哈希值难度匹配；以太坊中的挖矿计算也使用了哈希算法，其中的梅克尔 – 帕特里夏树同样也是一棵哈希树；其他的区块链系统也都会多多少少使用到各种哈希算法，因此可以说哈希算法贯穿到区块链系统的方方面面。

3.1.1 什么是哈希计算

密码学上的哈希计算方法一般需要具有以下的性质：

- ❑ 函数的输入可以是任意长的字符串；
- ❑ 函数的输出是固定长度的；
- ❑ 函数的计算过程是有效率的。

这个说法比较学术化，说白了，就是通过一个方法将一段任意输入的字符串计算出一个固定长度的值，相当于计算出一个身份证号。通过哈希算法计算出的结果，是无法再通过一个算法还原出原始数据的，即是单向的，因此适合用于一些身份验证的场合，同时由于哈希值能够起到一个类似于身份证号的作用，因此也可以用于判断数据的完整性，哪怕数据发生微小的变化，重新计算后的哈希值都会与之前的不一样。

一般来说，为了保证哈希函数在密码学上的安全性，必须满足以下 3 个条件。

1）抗冲突（collision-resistance）。简单来说，哈希函数抗冲突指的是不同的输入不能产生相同的输出。这就好像到电影院买票看电影，对于付出真金白银买了电影票的人，他们的座位号不能是一样的。同时必须说明的是，抗冲突并不是说不会有冲突，只不过找到有冲突的两个输入的代价很大，不可承受。这就好像暴力破解一个有效期为 20 年的密码，整个破解过程长达 30 年，虽然最后密码被破解了，但是由于密码有效期过了，所以也就失去了意义。

2）信息隐藏（information hiding）。这个特性是指如果知道了哈希函数的输出，不可能逆向推导出输入。这在密码学很好理解：即使敌人截获了公开信道（比如无线电波），获取了传送的哈希信息，敌人也不可能根据这段信息还原出明文。

3）可隐匿性（puzzle friendly）。如果有人希望哈希函数的输出是一个特定的值（意味着有人事先知道了哈希函数的输出结果），只要输入的部分足够随机，在足够合理的时间内都将不可能破解。这个特性主要是为了对付伪造和仿制。近来某位当红歌星的演唱会门票超贵，10 000 元一张。这就催生了假票行当：伪造个人演唱会的门票。这里门票是公开的，大家都知道长什么样，用什么材质，这相当于已知哈希函数的输出。可隐匿特性就是要做假票的明明知道输出长什么样，但不知道使用何种“原料”和“工艺”造出一模一样的票来。

注意 由于哈希算法的输出值是固定的，而原始数据的长度却是多种多样的，这就注定了在理论上存在不同的原始数据却输出同一种哈希值的可能，这种情况在原始数据的数量极其庞大的时候就会出现。比如，邮件系统的抗垃圾邮件算法，我们一般会对每一个邮件地址计算一个哈希值，存储为过滤库，可是全世界的邮件地址何其多，而且什么样格式都有，这个时候会对邮件地址进行多种哈希计算，将计算出来的多个值联合起来判断是否存在某个邮件地址，这也是布隆过滤器的基本原理，在比特币中就使用了布隆过滤器使 SPV 节点可以快速检索并返回相关数据。

3.1.2 哈希算法的种类

密码学中常用的哈希算法有 MD5、SHA1、SHA2、SHA256、SHA512、SHA3、RIPEMD160，下面简单介绍一下。

- MD5（Message Digest Algorithm 5）。MD5是输入不定长度信息，输出固定长度128bits的算法。经过程序流程，生成4个32位数据，最后联合起来成为一个128bits哈希。基本方式为求余、取余、调整长度、与链接变量进行循环运算，得出结果。MD5算法曾被广泛使用，然而目前该算法已被证明是一种不安全的算法。王晓云教授已经于2004年破解了MD5算法。
- SHA1。SHA1在许多安全协议中广为使用，包括TLS和SSL。2017年2月，Google宣布已攻破了SHA1，并准备在其Chrome浏览器产品中逐渐降低SHA1证书的安全指数，逐步停止对使用SHA1哈希算法证书的支持。
- SHA2。这是SHA算法家族的第二代，支持了更长的摘要信息输出，主要有SHA224、SHA256、SHA384和SHA512，数字后缀表示它们生成的哈希摘要结果长度。
- SHA3。看名称就知道，这是SHA算法家族的第三代，之前名为Keccak算法，SHA3并不是要取代SHA2，因为目前SHA2并没有出现明显的弱点。
- RIPEMD-160（RACE Integrity Primitives Evaluation Message Digest 160）RIPEMD160是一个160位加密哈希函数。它旨在替代128位哈希函数MD4、MD5和RIPEMD-128。

事实上，除了以上的算法，哈希算法还有很多种，有一些是不太讲究加密特性的，比如在负载均衡领域常用的一致性哈希算法，目的只是将服务器地址快速地计算出一个摘要值，而不是加密，因此会使用一些其他的快速哈希算法。

3.1.3　区块链中的哈希算法

1. 区块哈希

所谓区块哈希就是对区块头进行哈希计算，得出某个区块的哈希值，用这个哈希值可以唯一确定某一个区块，相当于给区块设定了一个身份证号，而区块与区块之间就是通过这个身份证号进行串联，从而形成了一个区块链的结构。这样的结构也是区块链数据难以篡改的技术基础之一。比如，一共有100个区块，如果要更改10号区块的数据，则11号就不能与10号连接，区块链就会断开，这样等于篡改无效了，而如果篡改了11号，就接着要篡改12号，以此类推，几乎就是牵一发动全身。如果区块链很长，那么要想更改之前的历史数据几乎就是不可能的了。从这个角度来看，哈希值相当于一个指针，传统的指针提供了一种获取信息的方法，而哈希指针则提供了一种检验信息是否被改编的方法，如果信息被篡改，那么其哈希值和哈希指针的值必定是不等的。

2. 梅克尔树

我们在第1章简单介绍过梅克尔树，梅克尔树在不同的区块链系统中有不同的细节，但本质是一样的，我们就以比特币中的梅克尔树来说明。比特币中的梅克尔树称为二叉梅

克尔树，每一个区块都有自己的梅克尔树，是通过将区块中的交易事务哈希值两两结对计算出新的哈希值，然后哈希值再两两结对进行哈希计算，递归循环，直到计算出最后一个根哈希值，这样的一棵树也称为哈希树。梅克尔树既能用于校验区块数据的完整性，也能对 SPV 钱包进行支付验证。

举一个生活中常见的例子，当我们签订一份 *n* 页的合同时，通常都会在每页合同上盖章，只不过每一页上的章都是一样的，这就给作弊留下了空间。如果我们稍微改变一下做法，给每一页合同盖一个数字印章，并且每一页上的数字印章是前一页数字印章和本页内容一起使用哈希算法生成的哈希值。例如：

1）合同第一页的数字印章是本页内容的哈希值，即第一页数字印章 =Hash（第一页内容）。

2）合同第二页的数字印章是第一页的数字印章及第二页内容加在一起后再哈希的值，第二页数字印章 =Hash（第一页的数字印章 + 第二页内容）。

3）合同第三页的数字印章是第二页的数字印章及第三页内容加在一起后再哈希后的值，即第三页数字印章 =Hash（第二页的数字印章 + 第三页内容）。

4）上述过程以此类推。

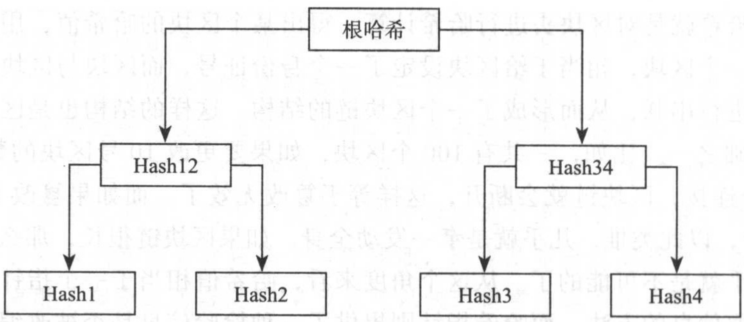
这样对第一页合同的篡改必然使其哈希值和第一页上的数字印章不符，且其后的 2，3，4，5，…，*n* 页也是如此；对第二页合同的篡改必然使其哈希值和第二页上的数字印章不符，且其后的 3，4，5，…，*n* 页也是如此。

从上面的例子，我们可以发现梅克尔树的优势：

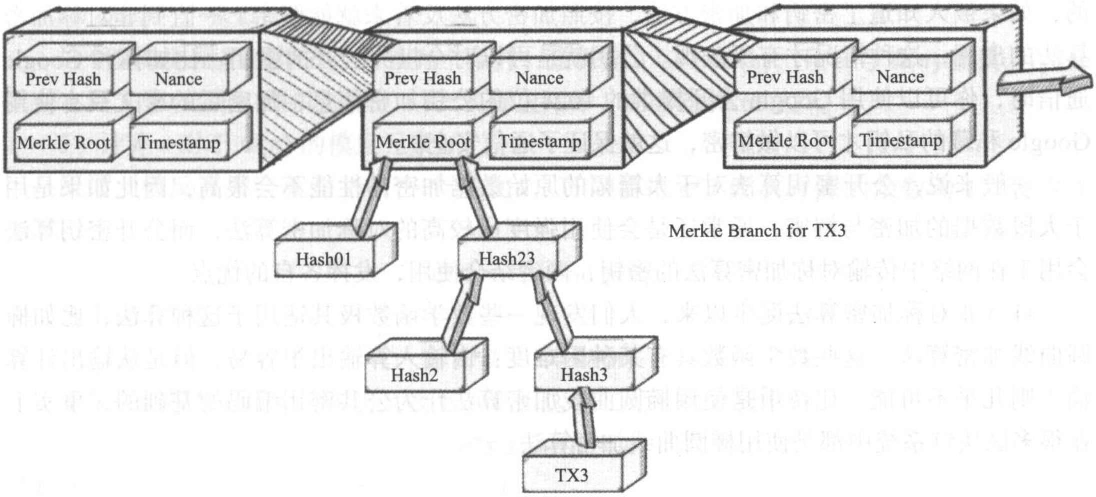
1）我们能知道信息是否被篡改；

2）我们还能知道是第几页或者第几块的信息被篡改了。

为了便于理解，我们看下梅克尔树的典型架构，如下图所示。



我们看到，首先这是一个树结构，在底部有 4 个哈希值，假设某个区块中一共有 4 条交易事务，那么每条交易事务都计算一个哈希值，分别对应这里的 Hash1 到 Hash4，然后再两两结对，再次计算哈希值，以此类推，直到计算出最后一个哈希值，也就是根哈希。这样的一棵树结构就称为梅克尔树（merkle tree），而这个根哈希就是梅克尔根（merkle root）。我们再来看一个示意图：



可以看到，每一个区块都是具有一棵梅克尔树结构的，同时可以发现，梅克尔树中的每一个节点都是一个哈希值，因此也可以称之为哈希树，而比特币中的梅克尔树是通过交易事务的哈希值两两哈希计算而成，所以这样梅克尔树称为二叉梅克尔树，那么这样的树结构有什么作用呢?

比特币是分布式的网络结构，当一个节点需要同步自己的区块链账本数据时，并没有一个明确的服务器来下载，而是通过与其他的节点进行通信实现的。在下载区块数据的时候，难免会有部分数据会损坏，对于这些一条条的交易事务，如何去校验有没有问题呢?这个时候，梅克尔树就能发挥作用了，由于哈希算法的特点，只要参与计算的数据发生一点点的变更，计算出的哈希值就会改变。我们以第一个示意图来说明，假设 A 通过 B 来同步区块数据，同步完成后，发现计算出的梅克尔根与 B 不一致，也就是有数据发生了损坏，此时先比较 Hash12 和 Hash34 哪个不一致，假如是 Hash12 不一致，则再比较 Hash1 和 Hash2 哪个不一致，如果是 Hash2 不一致，则只要重新下载交易事务 2 就行了。重新下载后，再计算出 Hash12 并与 Hash34 共同计算出新的梅克尔根比较，如果一致，则说明数据完整。我们发现，通过梅克尔树，可以很快找到出问题的数据块，而且本来一大块的区块数据可以被切分成小块处理。

3.2 公开密钥算法

3.2.1 两把钥匙：公钥和私钥

公钥和私钥是现代密码学分支非对称性加密里面的名词，对于一段需要保护的信息，通常使用公钥加密，用私钥解密，这种加密方法也称为公开密钥算法。

在谍战剧里，发电报那种一般都是使用对称加密算法。这种加密方式缺点是显而易见

的，如果被人知道了密钥和加密方法，按照加密方法反着来就能解密。一直到非对称加密算法的出现，这种情况才有所改观。公钥就是可以对全世界公开的密钥，比如你和Google通信时，你可以使用Google公开提供的1024位的公钥加密信息，加密后的密文只有使用Google私藏的私钥才可以做解密，这就保证了通信安全。

一般来说，公开密钥算法对于大篇幅的原始数据加密的性能不会很高，因此如果是用于大段数据的加密与解密，通常还是会使用强度比较高的对称加密算法，而公开密钥算法会用于在网络中传输对称加密算法的密钥，两者结合使用，发挥各自的优点。

自从非对称加密算法诞生以来，人们发现一些数学函数极其适用于这种算法，比如椭圆曲线加密算法。这些数学函数具有某种困难度：由输入算输出很容易，但是从输出计算输入则几乎不可能。比特币是使用椭圆曲线加密算法作为公共密钥编码的基础的，事实上在很多区块链系统中都是使用椭圆曲线加密算法。

3.2.2 RSA算法

RSA以它的三个发明者Ron Rivest、Adi Shamir和Leonard Adleman的名字首字母命名。RSA加密算法是最常见的非对称加密算法。它既能用于加密，也能用于数字签名，是目前最流行的公开密钥算法。RSA安全基于大质数分解的难度，RSA的公钥和私钥是一对大质数，从一个公钥和密文恢复明文的难度，等价于分解两个大质数之积，这是公认的数学难题。

RSA的安全基于大数的因子分解，但并没有从理论上证明破译RSA的难度与大数分解难度等价，RSA的重大缺陷是无法从理论上把握它的保密性能如何。只不过RSA从提出到现在20多年，经历了各种攻击的考验，被普遍认为是目前最优秀的公钥方案之一。RSA的缺点是：

- 产生密钥很麻烦，受限于质数产生的技术；
- 分组长度太大，运算代价高，速度慢。

我们通过一个例子来理解RSA算法。假设Alice要与Bob进行加密通信，她该怎么生成公钥和私钥呢？

1）选择两个质数。通常是随机选择两个不同的质数，我们不妨称为p和q，本例中Alice选择了61和53，当然实际应用中，这两个质数越大越好，这样就越难破解。

2）计算p和q的乘积n。Alice把61和53相乘：$n = 61 \times 53 = 3233$。

n的长度就是密钥长度，3233写成二进制是110010100001，一共有12位，所以这个密钥就是12位，实际应用中，RSA密钥一般是1024位，重要场合则为2048位，还是那句话，越长越好。

3）计算n的欧拉函数$\Phi(n)$。

根据公式：$\varphi(n) = (p-1)(q-1)$，Alice算出$\varphi(3233)$等于60×52，即3120，实际上就是两个质数分别减1后的乘积。

4）选择一个整数 e。

这个整数是随机选择的，并且有个条件，条件是 $1< e < \Phi(n)$，且 e 与 $\Phi(n)$ 互质。Alice 就在 1 到 3120 之间，随机选择了 17，实际应用中，常常选择 65 537。

5）计算 e 对于 $\Phi(n)$ 的模反元素 d。

所谓“模反元素”就是指有一个整数 d，可以使得 $e*d$ 被 $\varphi(n)$ 除的余数为 1，表达式如下：

$$e*d \equiv 1 \pmod{\varphi(n)}$$

这个式子等价于

$$e*d - 1 = k\varphi(n)$$

于是找到模反元素 d，实质上就是对下面这个二元一次方程求解：

$$e*x + \varphi(n)y = 1$$

已知 e=17，$\varphi(n)$=3120，则 $17x + 3120y = 1$。

这个方程可以用“扩展欧几里得算法”求解，此处省略具体过程。总之，Alice 算出一组整数解为 (x, y)=(2753, −15)，即 d=2753。

6）产生公钥和私钥。

将 n 和 e 封装成公钥，n 和 d 封装成私钥，在 Alice 的例子中，n=3233，e=17，d=2753，所以公钥就是（3233, 17），私钥就是（3233, 2753）。

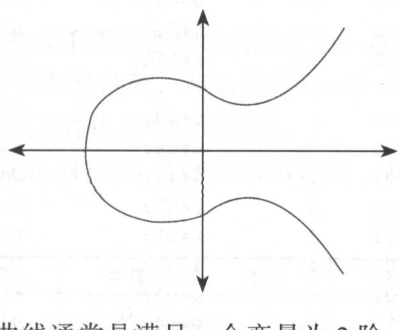
至此所有计算就完成了，可以看到 RSA 的算法过程其实还是很简单的，最关键的就是找到两个足够大的质数。

3.2.3　椭圆曲线密码算法

椭圆曲线是满足一个特殊方程的点集，注意，不要跟标准椭圆方程混淆，那根本就是两回事，看一个标准的椭圆曲线方程：

$$y^2=x^3+ax+b$$

在几何意义上，它通常是这样的一个图形，如下所示：



如上图所示，一个椭圆曲线通常是满足一个变量为 2 阶，另一个变量为 3 阶的二元方程。按照这样的定义，椭圆曲线是有很多种的，而椭圆曲线密码算法是基于椭圆曲线数学

的一种公钥密码算法，其主要的安全性在于利用了椭圆曲线离散对数问题的困难性。

在区块链中，常用的是ECDSA（椭圆曲线数字签名算法），这是利用椭圆曲线密码（ECC）对数字签名算法（DSA）的模拟。ECDSA于1999年成为ANSI标准，并于2000年成为IEEE和NIST（美国国家标准与技术研究院）标准。椭圆曲线密码算法实现了数据加解密、数字签名和身份认证等功能，该技术具有安全性高、生成公私钥方便、处理速度快和存储空间小等方面的优势。相对于RSA算法，在实际的开发使用中，椭圆曲线加密使用得更广泛，比如比特币就是使用了椭圆曲线中的SECP256k1，可以提供128位的安全保护。椭圆曲线具体的数学原理，其过程证明比较晦涩枯燥，这里不再赘述，感兴趣的朋友可以去查阅一些相关的数学资料。

3.3 编码/解码算法

众所周知，计算机存储和处理的都是二进制数据。为了简洁，实际上使用最多的是二进制的一个变种——十六进制。比如笔者的名字叫嘉文，中文拼音是jiawen（全小写），在计算机里存储的就是6A696177656E。很明显，人类容易记住jiawen，而其相应的十六进制代码6A696177656E就很考验人的记忆力了。同样，人类很难记住十六进制的数据，但如果是十六进制编码的文本字符串，就相对好记好读一些了。以下是一张ASCII码表的一部分。

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name Description
0	000	00	00000000	NUL	&#000;	*Null char*
1	001	01	00000001	SOH	&#001;	*Start of Heading*
2	002	02	00000010	STX	&#002;	*Start of Text*
3	003	03	00000011	ETX	&#003;	*End of Text*
4	004	04	00000100	EOT	&#004;	*End of Transmission*
5	005	05	00000101	ENQ	&#005;	*Enquiry*
6	006	06	00000110	ACK	&#006;	*Acknowledgment*
7	007	07	00000111	BEL	&#007;	*Bell*
8	010	08	00001000	BS	&#008;	*Back Space*
9	011	09	00001001	HT		*Horizontal Tab*
10	012	0A	00001010	LF
Line Feed
11	013	0B	00001011	VT	&#011;	*Vertical Tab*
...						
47	057	2F	00101111	/	/	*Slash or divide*
48	060	30	00110000	0	0	*Zero*
49	061	31	00110001	1	1	*One*
50	062	32	00110010	2	2	*Two*
51	063	33	00110011	3	3	*Three*
52	064	34	00110100	4	4	*Four*
53	065	35	00110101	5	5	*Five*
54	066	36	00110110	6	6	*Six*
55	067	37	00110111	7	7	*Seven*
56	070	38	00111000	8	8	*Eight*

```
57   071   39    00111001    9        &#57;         Nine
58   072   3A    00111010    :        &#58;         Colon
...
70   106   46    01000110    F        &#70;         Uppercase F
71   107   47    01000111    G        &#71;         Uppercase G
72   110   48    01001000    H        &#72;         Uppercase H
73   111   49    01001001    I        &#73;         Uppercase I
74   112   4A    01001010    J        &#74;         Uppercase J
75   113   4B    01001011    K        &#75;         Uppercase K
76   114   4C    01001100    L        &#76;         Uppercase L
77   115   4D    01001101    M        &#77;         Uppercase M
78   116   4E    01001110    N        &#78;         Uppercase N
79   117   4F    01001111    O        &#79;         Uppercase O
80   120   50    01010000    P        &#80;         Uppercase P
81   121   51    01010001    Q        &#81;         Uppercase Q
82   122   52    01010010    R        &#82;         Uppercase R
www.ascii-code.com 6A696177656E
```

十六进制的 07 是一个 Bell（响铃），如果试着用计算机程序去打印，结果是不可见，也不可理解的，只能听到一声铃声。但是文本字符串 "07" 则相对容易理解和记忆。上文提到过，比特币地址都是十六进制的数，不做转换，打印的话毫无意义，人类无法直观地辨识。大家可以想象一下查询自己的银行账户余额的场景：假如账户里只有 77 块钱了，查询结果打印的是大写字符 M（十进制的编码是 77）。我相信大部分用户都不知道那是 77 的意思。相对的，如果把数字 77 转换成文本 "77"（其十六进制编码是 3737）后再打印，对于显示在屏幕上的文本 77，用户就会理解了。总结一下：

	数字 7	字符 7	数字 77	字符 77
实际存储值（十六进制）	7	37	77	3737
打印到屏幕的结果	一声铃声	7	M	77

下面的几节将讨论用文本来表示十六进制数据的几种编码方式。

3.3.1　Base64

这是一种用 64 个字符来表示任意二进制数据的方法，通常 exe、jpg、pdf 等文件都是二进制文件，用文本编辑器打开都是乱码，那么就需要一个方法，可以将二进制编码成字符串的格式，这样可以将二进制文件用文本打开查看。

那么，既然是 Base64，就是通过 64 个字符来编码的，具体是哪 64 个字符呢？请见下表：

序号	字符	序号	字符	序号	字符	序号	字符
0	A	3	D	6	G	9	J
1	B	4	E	7	H	10	K
2	C	5	F	8	I	11	L

（续）

序号	字符	序号	字符	序号	字符	序号	字符
12	M	25	Z	38	m	51	z
13	N	26	a	39	n	52	0
14	O	27	b	40	o	53	1
15	P	28	c	41	p	54	2
16	Q	29	d	42	q	55	3
17	R	30	e	43	r	56	4
18	S	31	f	44	s	57	5
19	T	32	g	45	t	58	6
20	U	33	h	46	u	59	7
21	V	34	i	47	v	60	8
22	W	35	j	48	w	61	9
23	X	36	k	49	x	62	+
24	Y	37	l	50	y	63	/

Base64 编码主要用在传输、存储、表示二进制等领域，还可以用来加密。但是这种加密比较简单，只是一眼看上去不知道什么内容罢了，对应编码规则，可以很容易的解码，当然也可以对 Base64 的字符序列进行定制来进行加密，我们来看下 Base64 的编码过程。

首先，既然是使用上述 64 个字符的范围来表示的，那么要能够表示出 64 个字符的各种组合，得起码用 6 个 bit 才行，根据排列组合，6 个 bit 可以总共表示出 2^6 个组合的字符排列；针对一份需要转化的二进制文件，可以这样来处理，每 3 个字节一组，这样一共是 24bit，然后可以针对这个 24bit 再来划分，划分成每 6bit 一组，这样一共可以分成 4 组，则对照上表去查找对应的字符就可以了，这样就可以转换为 Base64 了，简单吧。

那么，如果在 3 个字节一组划分的时候，如果不是 3 的倍数怎么办呢？这样就需要使用 \x00 字节在末尾补足，再在编码的末尾加上 1 个或 2 个 = 号，表示补了多少字节。

由于标准的 Base64 编码后可能出现字符 + 和 /，在 URL 中就不能直接作为参数，所以又有一种 url safe 的 Base64 编码，其实就是把字符 + 和 / 分别变成 – 和 _。

根据这个原理，其实还是比较容易理解这种编码思想的，而且也可以看出，这种编码是可以逆向的，以 "yes" 这个字符串为例，它的 Base64 编码是 eWVz，大家可以自行尝试几个例子。

3.3.2 Base58

顾名思义，Base58 是基于 58 个字母和数字组成的，有了 Base64 的基础，我们就比较容易理解 Base58 了，实际上就是 Base64 的一个子集，相对于 Base64 来说，Base58 不包括以下 Base64 的字符：

- 数字 0

- ❑ 大写字母 O
- ❑ 大写字母 I
- ❑ 小写字母 l
- ❑ + 与 /

可以看出，小写 o 和大写 O 很容易和数字 0 混淆，小写 l 和大写 I 很容易和数字 1 混淆，Base58 就是 Base64 去除了几个看起来容易混淆的字符，以及容易导致转义的 / 和 +。Base58 的编码表如下：

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

必须注意，不同的应用实现使用的编码表内容是一样的，但是顺序可能不一样，比如：

1）比特币地址：123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz;

2）Ripple 地址：rpshnaf39wBUDNEGHJKLM4PQRST7VWXYZ2bcdeCg65jkm8oFqi1tuvAxyz。

接下来我们来了解一下 Base58Check，比特币使用的是改进版的 Base58 算法，是为了解决 Base58 编码的字符串没有完整性校验机制。在传播过程中，如果出现某些字符损坏或者遗漏，就没法检测出来了，所以使用了改进版的算法 Base58Check。

3.3.3　Base58Check

在二进制数据的传输过程中，为了防止数据传输的错误，保护数据安全，通常会加一个校验码，通过校验码的配合可以发现数据是否被破坏或者是否在发送时输入错误了。Base58Check 就是 Base58 加上校验码，或者可以说是 Base58 的一种编码形式，在比特币系统中生成钱包地址的时候就使用到了这种编码形式。我们知道，钱包地址是用来转账的，虽然 Base58 编码已经可以做到避免一些容易混淆的字符，但是还不能保证用户的误输入或者地址信息在传输过程中由于某种原因被损坏，这会给用户带来潜在的损失风险。

Base58Check 的编码方式，在我们第 1 章中介绍比特币地址的时候已经提到过，它的编码方式是这样的：进行编码前，在待编码的内容字符串中加入一个字节的版本信息，版本信息可以自行约定，比如比特币地址采用了 0x00 作为版本信息，然后再加入待编码内容字符串的哈希值，通常只要取得哈希值中的 4 个字节就可以了，加到一起后，然后再整体进行 Base58 编码。比特币地址的生成过程中，是将版本字节放在了头部，而将 4 个字节的哈希值放在了尾部，然后进行编码生成。这个原理还是很简单的，哈希算法具有先天的数据完整性检测能力，在这里我们又看到了哈希算法的又一个应用。

经过整体编码后的数据在传输过程中如果有发生损坏或者篡改，接收方在得到数据后，会对原始数据进行同样的校验码计算，并且和接收到的结果中的校验码进行比较。由于哈希算法的特点，只要原始数据有任何更改，计算出的哈希值都会发生变更，因此只要校验

码不一致就说明数据不是合法的。

3.4 应用场景

密码算法在区块链系统中的重要性，相当于整个体系的骨骼，如果没有骨骼会怎样？毫无疑问，整个大厦将会坍塌，我们来举一些例子，看看都起哪些作用。

（1）账户地址生成

这个其实就是对公开密钥算法的巧妙使用，首先生成一对密钥，即私钥和公钥，由于公钥是可以公开的，因此可以作为自己对外的一个账号，而又由于公钥必须和对应的私钥匹配才能验证通过，因此这种方式生成的地址，先天就具备可验证性。

（2）价值转移保卫

我们不展开对价值转移本身经济意义的论述，就说实现方式，这又是公开密钥算法的一个用武之地了。无论是比特币、以太坊、超级账本 Fabric 还是其他区块链系统，要想在一个分布式的公网上发送一笔代表价值的数据（比如数字货币、证券、资产所有权等），必须解决掉两个基本的问题：

1）证明这笔数据确实是发出者的，不是篡改或者伪装的；

2）确保只有接收者才能解码这笔携带价值的数据。

毫无疑问，这两点要求，可以通过公开密钥算法完美地解决，发送者使用自己的私钥进行签名，相当于盖上了自己的公章，接收者可以使用发送者公开的那个公钥进行身份验证以确保无误。发送者不但使用了自己的私钥签名，还使用了接收者的公钥进行了一段关键的加密，只有接收者使用自己的私钥才能解密这个公钥，因此就能保证不被别人截获，或者说即使被截获了也没关系，因为别人没有对应的私钥来解码。

（3）完整性证明

这个领域就是我们哈希算法的战场，我们在上述内容中也有介绍，在节点同步区块数据时，通过构建的交易哈希树来验证数据是否一致。

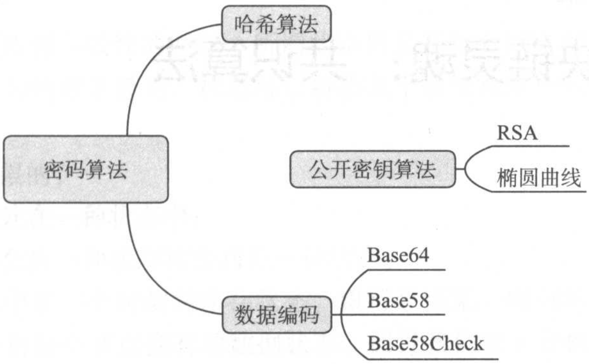
（4）零知识证明

要想证明自己拥有某笔资产或者拥有某个能力，或者更直接地说，要想证明自己具备对区块链上某一笔交易的所有权，应该怎么办？通常的思路自然是提交自己的密码，看能不能解锁匹配，可是这样的话，密码就泄露了，不但密码会泄露，交易内容也可能就此公开了，隐私全没了，那该怎么办？毫无疑问，在这个场合，密码算法起到了非常大的作用，只要解码一段与交易内容相关但是又不泄露真正交易内容的编码，能够解码成功就能证明所有权了。

密码算法在区块链中的应用是非常重要的，以上只是列举了一些常见的应用点，在实际应用中，还有很多地方是有非常巧妙的应用的，老实说，直到现在，笔者也仍然惊叹于比特币一开始通过公钥生成钱包地址的做法，虽然现在看起来已经没什么大不了的了，然

而换做当年，扪心自问，本人还真很难设计出如此绝妙的主意，传统的技术，加上创新的用法，可以产生非常大的威力。

3.5 知识点导图



Chapter 4 第 4 章

区块链灵魂：共识算法

4.1 分布式系统的一致性

所谓一致性，就是指数据要完整、要同步。比如，我们在网上下了个订单，付了款，商城系统会记录下这些数据，之后无论我们在哪里访问自己的订单，都能看到同样的结果，即便商城系统出了故障，那一般也会返回一个错误提示而不是给我们一个不一样的结果。再比如银行，我们存了一笔钱进去，无论到哪里，我们查询银行账户的时候金额也不会变。也就是说，在这些系统中，数据的结果总是一致而同步的，即便这些系统的服务器不止一台，但都属于同一个中心集群，在内部是可以高效一致同步的。

区块链系统本质就是一个分布式应用软件。分布式系统的首要问题就是如何解决一致性的问题，也就是如何在多个独立的节点之间达成共识。要注意的是，这里说的是要达成一致，而没有说保证一定要结果正确，比如所有的节点都达成失败状态也是一种一致，说白了就是要成为一致行动人。如果是中心化的场景，达成一致几乎没有任何问题，但分布式环境并不是那么理想的，比如节点之间的通信可能是不可靠的、会有延迟、会有故障，甚至节点直接宕机。而在无数个节点之间，如果采用同步的方式来调用，那等于失去了分布式的优点，因为绝对理想的一致性，代价通常是很大的。这样一个模型，通常要求不发生任何故障，所有节点之间的通信没有任何时差，这几乎就等于是一台计算机了，这样的强一致性往往意味着性能较弱。

历史经验表明，多路处理器和分布式系统中的一致性问题是非常难以解决的。难点在于以下几个方面。

1）分布式系统本身可能出故障。

2）分布式系统之间的通信可能有故障，或者有巨大的延迟。

3）分布式系统运行的速度可能大不相同，有的系统运行很快，而有些则很慢。

但是，一致性是设计分布式系统时必须考虑的问题。一致性问题历史悠久，而且臭名昭著。传统处理一致性问题的方法有两段式提交、令牌环、时间戳等，计算机专业的读者应该有所耳闻。本章将集中讨论与区块链相关的一致性问题和算法。

4.1.1　一致性问题

我们用状态机来解释一致性的问题。所谓状态机是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型，状态可以特指某个系统在某一时刻的数据变量，它具备以下几个特点：

- 状态总数是有限的；
- 任一时刻，只处在一种状态中；
- 某种条件下，会从一种状态转变到另一种状态。

比如进销存系统中某一个时刻的库存状态，银行系统某一时刻的账户状态等，对于分布式系统来说，就是指每个节点拥有的数据状态。假设我们有 n 台机器，位于不同位置的机器之间通过网络协同工作，所有机器的初始状态是一模一样的。给它们一组相同的指令，我们希望看到相同的输出结果，而且希望看到状态的变化也是一样的。比如机器甲的状态是从状态 A 到 B 再到 C，而如果机器乙的状态是由 A 直接到 C，这种情况就是不一致的。

总而言之，一致性要求分布式系统中每个节点产生同样的结果或者具备同样的状态，看起来就好像是一台机器一样，前提是没有一个中心服务器作为调度员，这对于分布在互联网上、不在同一个机房内、不属于同一个管理者的分布式系统来说，难度是很大的。出于系统的可用性考虑，对于分布式系统来说，我们一般希望具备以下能力。

1）分布式系统作为一个逻辑整体，不应该返回错误的结果。

2）只要系统里的大部分机器工作正常，整个分布式系统就能有效运行，这也是分布式系统应用的一个优点，抵抗单点故障。

3）系统的性能是可以横向扩展的，对于分布式系统来说，木桶原理不起作用。

4）分布式系统必须是异步的，也就是说每个节点可以按照自己的时序独立工作，没有全序的时间顺序。

要达到这些要求，可是不容易呢！从生活中我们也可以发现，即便有统一的命令指挥，尚且不一定能完全做到整齐划一，何况是没有这么一个指挥员呢！在互联网的场景中，任意一个节点的状态，我们都是没法去强力管控的，比如比特币节点，谁能控制网络中的那些节点呢！可能就是关闭了、断网了，甚至是一个恶意伪装的节点。一切看起来似乎无解。然而实际上，很多时候我们对一致性的要求并没有那么迫切，在一定的约束下，可以实现所谓的最终一致性，也就是说在某个时刻系统达到了一致的状态。这个节点现在断网了，没问题，等恢复后跟上，通过其他节点来同步自己的数据；那个节点宕机了，也没问题，恢复后跟上。只要整个网络中绝大部分的节点都是正常工作的，整个系统总能在未来的某

一个时刻达成数据状态的一致。

4.1.2 两个原理：FLP 与 CAP

1. FLP 定理

叫 FLP 是因为提出该定理的论文是由 Fischer、Lynch 和 Patterson 三位作者在 1985 年发表的，取了各自名字的首字母作为定理名称。看下定义：在网络可靠、存在节点失效（即使只有一个）的最小化异步模型系统中，不存在一个可以解决一致性问题的确定性算法。在这个原理的前提下，也告诉人们：**不要浪费时间去为异步分布式系统设计在任意场景下都能实现共识的算法，在允许节点失效的情况下，纯粹异步系统无法确保一致性在有限时间内完成**。

这个其实也很好理解，比如三个人在不同房间回答问题，虽然三个人彼此之间是可以通过电话沟通的，但是经常会有人时不时地开小差，比如 Alice 和 Bob 都回答了某个问题，Lily 收到了两者的回答结果，然后玩游戏去了，忘了回复，则三个人永远无法在有限时间内获得最终一致的答复。这个定理在理论上证明了此路不通，也就节省了后来者的研究时间。

2. CAP 定理

CAP 定理最早是由 Eric Brewer 在 2000 年 ACM 组织的一个研讨会上提出猜想，后来 Lynch 等人进行了证明。我们还是先来看下定义：分布式计算系统不可能同时确保**一致性**、**可用性**和**分区容错性**，这三者不可兼得。通过定义我们可以知道，这是一个典型的不可能三角。那么这三个术语具体是什么意思呢？含义如下。

- 一致性（consistency）：所有节点在同一时刻能够看到同样的数据，即“强一致性”。
- 可用性（availability）：确保每个请求都可以收到确定其是否成功的响应，并且是在有限的时间内。
- 分区容错性（partition tolerance）：因为网络故障导致的系统分区不影响系统正常运行，比如 1 号模块和 2 号模块不能使用了，但是 3 号和 4 号依然能提供服务。

直觉上的论证很简单：如果网络分成了两半，我在一半的网络中给 A 发送了 10 个币，在另外一半的网络中给 B 发送了 10 个币，那么要么系统不可用，因为其中一笔交易或者全部两笔都不会被处理，要么系统会变得没有一致性，因为一半的网络会完成第一笔交易，而另外一半网络会完成第二笔交易。

既然不能同时满足，那么如果弱化对某个特性的支持呢？

（1）弱化一致性

比如软件升级新版本后，过一段时间其他人才更新成功；再如网站更新内容后，浏览器也是刷新后才显示更新内容。很多时候对于实时的强一致性并没有很高的要求，生活中也有这样的例子：如果事情不那么紧急，就会发个短消息或者发个邮件，等对方看到了再

处理；如果紧急的话，就直接打电话，所以说电话是一种强连接方式，不过很多朋友肯定不喜欢时不时地就有电话打进来吧。

（2）弱化可用性

有些场合对一致性非常敏感，比如银行取款机，一旦系统故障就会拒绝服务，再如飞机上的控制系统，这个时候如果不能实时处理，那可是要命的。对计算机使用比较熟悉的朋友都知道，很多服务器操作系统都是不带图形界面的，只能靠命令行来处理，因为服务器要提高性能，保持可靠，会尽量避免加载不必要的模块，这也是一个牺牲可用性的例子。

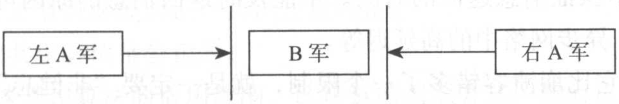
（3）弱化分区容错性

对于分布式系统来说，分区容错是必然的，幸运的是这种情况出现的概率并不是很大。如果真的是大规模的服务不可用，那无论是什么样的系统都是不能正常工作的。

计算机系统的设计，有时候跟生活中的场景是很类似的，你不得不做出一些妥协以便保证自己最想要得到的结果。区块链系统中，尤其是公有链系统，使用各种共识算法，优先的目的就是要保证整个系统的容错能力，这也是设计为分布式或者去中心结构的目的之一。

4.1.3　拜占庭将军问题

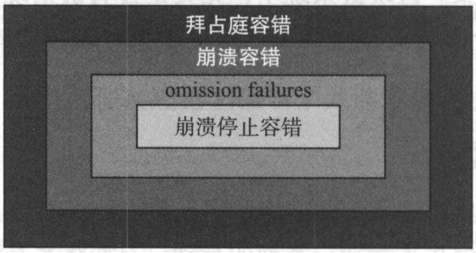
拜占庭将军问题的经典描述是：拜占庭的军队是由小分队组成的，每个小分队由一个将军指挥，将军们通过传令兵来策划一系列的行动。有些将军是叛徒，他们会有意地妨碍忠诚的将军达成一致的计划。这个问题的目标是使忠诚的将军达成一致的计划，即使背叛的将军一直在诱使他们采用糟糕的计划。已经证明，如果背叛的将军超过了将军总数的 1/3，达成上述目标是不可能的。特别要注意的是，要把拜占庭将军问题和两军问题区分开。两军问题的模型要比拜占庭将军问题简单，并且设立的前提场景也有差别，我们来看一幅示意图：



如上图所示，在此问题模型中，假设有两支对抗的军队（一支为 A 军，一支为 B 军），这也就是所谓的两军。A 军被 B 军隔开为两个部分，分别是左 A 军和右 A 军。从战斗力来说，A 军的两个部分必须同时合力进攻才能打败 B 军，这就要求 A 军的左右两支分队必须要协商好进攻时间和一些进攻的其他约定，协商就意味着要通信，通过两边的互相通信来保持进攻指令的一致性。那么问题来了，左右两边的 A 军要互相通信，就必须经过 B 军的区域，这就很难保证通信是畅通的，两边必须要不断发送回执来确认对方是否收到了消息，很显然，从理论上来讲，任何一次的回执都没法真正确认双方的消息接收是一致的。比如左 A 军发送了消息给右 A 军，右 A 军接收到了并且发送了确认回执给左 A 军，可是确认回执被 B 军阻截了，此时左 A 军无法知道右 A 军到底收到消息没有，即便右 A 军的回执成

功到达了左 A 军，可是若没有左 A 军的回执（左 A 军的回执也可能被 B 军阻截），右 A 军同样无法确认左 A 军到底收到回执没有。按照这种确认模式，只要有 B 军的阻截存在，左右两边 A 军就没法在理论上保证总是能达成一致的消息确认。

我们可以看到，两军问题的关键点在于：两点之间的信道传输不可靠。我们日常使用的大多数网络通信软件（支付、聊天、发送邮件等）其实都会面临这样的问题，通信过程发生在互联网，谁也没法保证中间经过的“B 军”是可靠的，一般也只能通过有限次数的双方回执来确认消息的到达，这也是一个不得已的折中方案。值得注意的是，在这个问题模型中，并没有去假设中间是否存在故意破坏者，也就是在两军的通信过程中，不考虑某一方可能叛变的情况。回到拜占庭将军问题，其考虑的主要问题在于通信的各方（不一定是两军，也可能是多军）存在故意破坏者或叛徒的情况下，大家如何来保持正确的一致性的问题。



拜占庭将军问题的复杂性，可以用计算机容错学里的概念来表述。

1）拜占庭容错：这是最难处理的情况，指的是有一个节点压根就不按照程序逻辑执行，对它的调用会返回随意或者混乱的结果。要解决拜占庭式故障需要有同步网络，并且故障节点必须小于 1/3，通常只有某些特定领域才会考虑这种情况，通过高冗余来消除故障。

2）崩溃容错：它比拜占庭容错多了一个限制，那就是节点总是按照程序逻辑执行，结果是正确的，但是不保证消息返回的时间。不能及时返回消息的原因可能是节点崩溃后重启了、网络中断了、异步网络中的高延迟等。

3）遗漏容错：它比崩溃容错多了一个限制，就是一定要“非健忘”。非健忘是指这个节点崩溃之前能把状态完整地保存在持久存储上，启动之后可以再次按照以前的状态继续执行和通信。比如最基本版本的 Paxos，它要求节点必须把投票的号码记录到持久存储中，一旦崩溃，修复之后必须继续记住之前的投票号码。

4）崩溃停止容错：它比遗漏容错多了一个故障发生后要停止响应的要求。简单讲，一旦发生故障，这个节点就不会再和其他节点有任何交互，就像它的名字描述的那样，崩溃并且停止。

4.1.4 共识算法的目的

在有错误的进程存在并且有可能出现网络分区的情况下，FLP 定理堵死了我们在传统

计算机算法体系下提出解决方案的可能性。计算机科学家就想，如果我们把 FLP 定理的设定放松一点，问题是否有解呢？由社会学和博弈论中得到启发，科学家尝试引入了以下机制。

1）**激励机制**（incentive）。比如，在拜占庭将军问题中给忠诚的将军以奖励。当背叛的将军发现背叛行为没有任何收益的时候，他们还有背叛的动机吗？这里引进了博弈论的概念：我们不再把节点或者说将军分成公正 / 恶意（忠诚 / 背叛）两方，认为每一个节点的行为是由激励机制决定的。正如两千年前中国诸子百家热烈争论的话题：人之初，性本善焉，性本恶焉？我们认为，人之初，性无善无恶。性的善恶由后天的激励机制决定。如果激励机制设置得当，考虑到每个节点都有最大化自己利益的倾向，大部分的节点都会遵守规则，成为公正的节点。

2）**随机性**（randomness）。在拜占庭将军问题中，决定下一步行动需要将军们协调一致，确定统一的下一步计划。在存在背叛将军的条件下，忠诚的将军的判断可能被误导。在传统的中心化系统中，由权威性大的将军做决定。在去中心化的系统中，研究者提出一种设想：是否能在所有的将军中，随机地指定一名将军作决定呢？这个有点异想天开的设想为解决拜占庭将军问题打开了一扇门。根据什么规则指定做决定的将军呢？对应到金融系统里，就是如何决定谁有记账权。

1）**根据每个节点（将军）的计算力（computing power）来决定**。谁的计算力强，解开某个谜题，就可以获得记账权（在拜占庭将军问题里是指挥权）。这是比特币里用的 **PoW 共识协议**。

2）**根据每个节点（将军）具有的资源（stake）来决定**。所用到的资源不能被垄断，谁投入的资源多，谁就可以获得记账权。这是 **PoS 共识协议**。

出于上面的考虑，科学家引入共识算法，试图解决拜占庭将军问题。分布式共识协议具有以下两点属性：

1）如果所有公正节点达成共识，共识过程终止；

2）最后达成的共识必须是公正的。

下面我们来谈谈共识算法的适用范围。区块链的组织方式一般有以下 3 种。

1）私有链：封闭生态的存储网络，所有节点都是可信任的，如某大型集团内部的多数公司。

2）联盟链：半封闭生态的交易网络，存在对等的不信任节点，如行业内部的公司 A、B、C 等。

3）公有链：开放生态的交易网络，即所有人都可以参与交易，没有任何限制和资格审核。

由于私有链是封闭生态的存储网络，因此使用传统分布式一致性模型应该是最优的；由于联盟行业链的半封闭、半开放特性，使用 Delegated Proof of ××× 是最优的；对于公有链，PoW 应该是最优的选择。

常见共识算法一览表：

共识算法	应用
PoW	比特币、莱特币，以及以太坊前 3 个阶段：Frontier（前沿）、Homestead（家园）、Metropolis（大都会）
PoS	PeerCoin、NXT，以及以太坊的第 4 个阶段，即 Serenity（宁静）
DPoS	BitShare
Paxos	Google Chubby、ZooKeeper
PBFT	Hyperledger Fabric
Raft	etcd

4.2 Paxos 算法

首先，Paxos 算法解决的是非拜占庭将军问题，也就是说仅仅是指分布式系统中的节点存在故障，但是不存在恶意节点的场景，在这种情况下如何达成共识。

1998 年 Lamport 提出 Paxos 算法，后续又增添多个改进版本的 Paxos，形成 Paxos 协议家族。Paxos 协议家族有一个共同的特点就是不易于工程实现，Google 的分布式锁系统 Chubby 作为 Paxos 实现曾经遭遇到很多坑。

除了经典 Paxos（又名 Basic Paxos），以下均为 Paxos 的变种，基于 CAP 定律，侧重了不同方向。

- Cheap Paxos
- Egalitarian Paxos
- Fast Paxos
- Multi-Paxos
- Byzanetine Paxos

Paxos 算法实在是太晦涩难懂，上面所列的 Paxos 算法分支就不详细介绍了。如果要想了解一下经典 Paxos 算法的最初描述，可以去看一下 Lamport 的论文《 Paxos Made Simple 》，在这个算法模型中，使用到了如下的角色：

角色	描述
提议议案人	由提议人提出，审批人进行审核，审核内容主要包括议案的编号和内容
提议人	议案的提出者，并且接受审批人的审核意见
审批人	对提议人提出的议案进行审核并返回意见结果
执行人	当议案成为决议后，通知所有执行人

看到这些角色，有没有觉得很像现代的议会制度。Paxos 正是这样的一个模型，当然在计算机中这些所谓的“人”一般就是指节点，这些角色可以是不同的服务节点也可以是同一个服务节点兼任。提案发出后，就要争取大多数的投票支持，当超过一半支持的时候，发送一半结果给所有人进行确认，也就是说 Paxos 能保证在超过一半的正常节点存在时，系统达成共识。提案过程还可以划分不同的场景，如下所示：

（1）单个提案者 + 多个接收者

这种情况下，一致性容易达成，或者说肯定能达成，因为只有一个提案，要么达成，要么否决或者失败。但是这种情况下，这个唯一的提案者如果出故障，则整个系统就失效了。

（2）多个提案者 + 单个接收者

这种情况下也容易达成共识，对于接收者，选择一个作为决议即可，当然这种情况也属于单点故障结构。

（3）多个提案者 + 多个接收者

这种情况，首先是避免了单点故障，但是问题也变得复杂了，既然提案和接收者都有多个，那以哪个为准呢？并没有特别玄妙的办法，既然多个在一起不好解决，那还是得回到单个提案者上去，只不过增加个规则选出那么一个单个提案者来，大致可以有如下的两个方案：

1）与第一种情况靠近，也就是想个办法选出一个提案者出来，约定在某一个时间段内，只允许一个提案通过，可以设置一些竞争规则或者按照一个时间序列的排列选择，总之最后会选出一个提案者。

2）与第二种情况靠近，允许有多个提案者，但是当节点收到多份提案后，通过某个规则选出一份提案，也就是仍然保持只接收一份，规则可以有各种，比如根据提案序号排列或者根据提案时间等。

实际上，在网络中，类似比特币这种，必然是属于多对多的这种情况，发送转账交易的节点不止一个，矿工不止一个，接收区块进行验证的节点当然也不止一个，Paxos 中为了解决这样的问题，引入了称为“两阶段提交”的方案。所谓两阶段，就是“准备”和“提交”两个阶段：**准备阶段解决大家对哪个提案进行投票的问题，提交阶段解决确认最终值的问题**。上述这个过程中，可能会一直有新的提案出现，因此类似于比特币一样，分隔一下时间，比如每隔 10 分钟打包一次，而打包者只能有一个。

在提交阶段，如果一个提案者在准备阶段接收到大多数节点的回复，则会发出确认消息，如果再次收到大多数的回复，则保持原先的提案编号和内容；如果收到的消息中有更新的提案，则替换为更新的提案内容；如果没有收到大多数的回复，则再次发出请求，等待其他节点的回复确认。当接收者发现提案号与自己目前保留的一致，则对提案进行确认。

就个人的理解，这种做法如果是在一个相对私有的环境中或者网络环境比较好的情况下，效果会比较明显，实际上，所谓的收到大多数的回应，这也是节点自身的一个评估，因为节点并没有更好的办法去判断，到底算不算是大多数了，尤其是节点总数还不固定的情况下。

4.3 Raft 算法

由于 Paxos 太难懂、太难以实现，Raft 算法应运而生。其目的是在可靠性不输于 Paxos

的情况下，尽可能简单易懂。斯坦福大学的 Diego Ongaro 和 John Ousterhout 以易理解为目标，重新设计了一个分布式一致性算法 Raft，并于 2013 年底公开发布。Raft 既明确定义了算法中每个环节的细节，也考虑到了整个算法的简单性与完整性。与 Paxos 相比，Raft 更适合用来学习以及做工程实现。下面，笔者将以通俗易懂的方式来描述这个过程。

百花村村长一人负责对外事务。比如，县和乡两级的公文来往，公粮征收，工务摊派，税收等。

Raft 是一个强 Leader 的共识协议。我们想象百花村是一个服务器集群，而这个集群的 Leader 就是村长，村里的每户人家（follower）对应一个服务器，每户人家都保存了一个数据副本。所有的数据副本都必须保证一致性。即上级官员下到村里视察时，从每户人家获得的信息应该是一样的。

百花村村长通过村户选举产生。谁得的票数多（简单多数）谁就当选村长。村长有任期概念（term）。任期是一直向上增长的：1，2，3，…，n，n+1，…。

这里要处理的是平票（split vote）的情况。在平票的情况下，该村村长选举失败，每户人家被分配不同的睡眠值。在睡眠期间的村户不能发起选举，但是可以投票。而且只有选举权，但是没有被选举权。第一个走出睡眠期的村户发起新任期的选举。由于每户人家有不同长度的睡眠期，这保证了选举一定会选出一个村长，而不会僵持不下，不会出现每次选举都平票的情况。一旦村长产生，任何针对百花村的“写”（比如政府政策宣示，普法教育）必须经过村长。

村长每天都要在村里转一圈，让所有人都看见。表明村长身体健康，足以处理公务。

村长选举出来后，要防止村长发生“故障”，必须定期检测村长是否失效。一旦发现村长发生“故障”，就要重新选举。

村长接收到上级命令，该命令数据处于未提交状态（uncommitted），接着村长会并发向所有村户发送命令，复制数据并等待接收响应，确保至少超过半数村户接收到数据后再向上级确认数据已接收（命令已执行）。一旦向上级发出数据接收 Ack 响应后，表明此时数据状态进入“已提交”（committed），村长再向村户发通知告知该数据状态已提交（即命令已执行）。

下面我们来测试各种异常情况。

（1）异常情况 1

上级命令到达前，村长挂了。这个很简单，重新选举村长。上级命令以及来自外面的请求会自动过时失效，他们会重发命令和请求。

（2）异常情况 2

村长接到上级命令，还没有来得及传达到各村户就挂了。这个和异常情况 1 类似，重新选举村长。上级命令以及来自外面的请求会自动过时失效。他们会重发命令和请求。

（3）异常情况 3

村长接到上级命令，已传达到各村户，但是各村户尚未执行命令，村长就挂了。这种异常情况下，重新选举村长。新村长选出后，由于已收到命令，就可以等待各村户执行命

令（也就是 Commit 数据）。上级命令以及来自外面的请求会自动过时失效。有可能，他们会重发命令和请求。Raft 要求外部的请求可以自动去除重复。

（4）异常情况 4

村长接到上级命令，已传达到各村户，各村户执行了命令，但是村长并没有收到通知，就在这时候村长挂了。这种情况类似上一种情况，新村长选出后，即可等待通知，完成剩下的任务。外部也会接到通知命令（已完成）。

（5）异常情况 5

在命令执行过程中，村长身体不适，不能处理公务。因为百花村没有收到村长的“心跳”，百花村的村户就会自动选举（当前任期 +1）任村长。这个时候就出现 2 个村长。这个时候新村长就会接过老村长角色，继续执行命令。即使原村长身体康复，也将成为普通村户。

4.4 PBFT 算法

1999 年 Castro 和 Liskov 提出的 PBFT（Practical Byzantine Fault Tolerance）是第一个得到广泛应用的 BFT 算法。在 PBFT 算法中，至多可以容忍不超过系统全部节点数量的 1/3 的拜占庭节点“背叛”，即如果有超过 2/3 的节点正常，整个系统就可以正常工作。早期的拜占庭容错算法或者基于同步系统的假设，或者由于性能太低而不能在实际系统中运作。PBFT 算法解决了原始拜占庭容错算法效率不高的问题，将算法复杂度由指数级降低到多项式级，使得拜占庭容错算法在实际系统应用中变得可行。也许就是出于效率的考虑，央行推出的区块链数字票据交易平台用的就是优化后的 PBFT 算法。腾讯的区块链用的也是 PBFT。

在 PBFT 算法中，每个副本有 3 个状态：pre-prepare、prepared 和 commited。消息也有 3 种：pre-prepare、prepare 和 committed。收到 pre-prepare 消息并且接受就进入 prepared 状态。收到 commit 消息并且接受就进入 Committed 状态。下面以一个有 4 个节点 / 拷贝的例子说明，这个网络内，仅允许 1 个拜占庭节点（此处设 f=1）。

百花村小学举行百米赛跑比赛，3 年级第一组的选手只有 4 个人：Alice、Bob、Cathy 和 David（简称 A、B、C、D）。为了节省钱，比赛并没有请裁判，而是在 4 个选手中随机挑出一个做裁判，假设是 Alice。众所周知，百米跑的口令是：“各就各位，预备，跑！”

这里“各就各位”就是 pre-prepare 消息，选手接受了命令就会脚踩进助跑器，而这一动作被其他选手看到，就会认为该选手进入了 prepared 状态。相当于发了一个 prepare 消息给其他选手。同理，预备就是 prepare 消息，选手接受了就是双手撑起，身子呈弓形，而这一动作被其他选手看到，就会认为该选手进入了 committed 状态。

1）假设 A 是公正的。Alice 得到老师示意，3 年级第一组准备比赛。Alice 就喊：“各就各位！”

老师的示意相当于一个外部消息请求。Alice 收到这个消息，给消息编一个号，比如编

为 030101 号。必须编号，因为比赛有一个规则（假想），连续 4 次起跑失败，整个组都被淘汰。B、C、D 同学收到口令后，如果认为命令无误，便都把脚踩进助跑器（拜占庭的那个人例外）。而这一个动作又相当于互相广播了一个 prepare 消息。A、B、C、D 选手互相看到对方的动作，如果确认多于 f 个人（由于此处 f=1，所以至少是 2 个人）的状态和自己应有的状态相同，则认为大家进入 prepared 状态。选手会将自己收到的 pre-prepare 和发送的 prepare 信息记录下来。

2）假设 A 是公正的。Alice 看到至少 2 个人进入 prepare 状态，Alice 就接着喊："预备，跑！"。

3）接下来发生的事类似上一步：B、C、D 同学收到口令后（相当于收到 commit 消息），如果认为命令无误，便都双手撑起，身子呈弓形（拜占庭的那个人例外）。而这一个动作又相当于互相广播了一个 commit 消息。A、B、C、D 选手互相看到对方的动作，如果确认多于 f 个人（由于此处 f=1，所以至少是 2 个人）的状态和自己应有的状态相同，则认为大家进入 committed 状态。当大家都确认进入 Committed 状态后，就可以起跑了！

4）假设 A 是不公正的。A 就会被换掉，重新选一个选手 B 发令。

这时候，由于所有选手都记录了自己的状态和接受 / 发送的信息。那些换掉前已经是 Committed 状态的选手，开始广播 commit 消息，如果确认多于 f 个人（由于此处 f=1，所以至少是 2 个人）的状态和自己应有的状态相同，则认为大家进入 committed 状态。而对于换掉前是 prepared 和 pre-prepare 状态的选手，则完全作废以前的命令和状态，重新开始。

PBFT 算法的主要优点如下。

- PBFT 算法共识各节点由业务的参与方或者监管方组成，安全性与稳定性由业务相关方保证。
- 共识的时延大约在 2 ～ 5 秒，基本达到商用实时处理的要求。
- 共识效率高，可满足高频交易量的需求。

因为非常适合联盟链的应用场景，PBFT 及其改进算法因此成为目前使用最多的联盟链共识算法。改进主要集中在：①修改底层网络拓扑的要求，使用 P2P 网络；②可以动态地调整节点数量；③减少协议使用的消息数量等。

不过 PBFT 仍然是依靠法定多数（quorum)，一个节点一票，少数服从多数的方式，实现了拜占庭容错。对于联盟链而言，这个前提没问题，甚至是优点所在。但是在公有链中，就有很大的问题。

4.5 工作量证明——PoW

工作量证明（Proof of Work，以下简称 PoW）机制随着比特币的流行而广为人知。PoW 协议简述如下：

1）向所有的节点广播新的交易；

2）每个节点把收到的交易放进块中；

3）在每一轮中，一个被随机选中的节点广播它所保有的块；

4）其他节点在验证块中的所有的交易正确无误后接受该区块；

5）其他节点将该区块的哈希值放入下一个它们创建的区块中，表示它们承认这个区块的正确性。

节点们总是认为最长的链为合法的链，并努力去扩大这条链。如果两个节点同时广播各自挖出的区块，其他节点以自己最先收到的区块为准开始挖矿，但同时会保留另一个区块。所以就会出现一些节点先收到 A 的区块并在其上开始挖矿，同时保留着 B 的区块以防止 B 的区块所在的分支日后成为较长的分支。直到其中某个分支在下一个工作量证明中变得更长，之前那些在另一条分支上工作的节点就会转向这条更长的链。

平均每 10 分钟有一个节点找到一个区块。如果两个节点在同一个时间找到区块，那么网络将根据后续节点的决定来确定以哪个区块构建总账。从统计学角度讲，一笔交易在 6 个区块（约 1 个小时）后被认为是明确确认且不可逆的。然而，核心开发者认为，需要 120 个区块（约一天），才能充分保护网络不受来自潜在更长的、已将新产生的币花掉的区块链的威胁。

生物学上有一个原理叫作“不利原理”（handicap principle），该原理可以帮助我们解释工作量证明的过程。这个原理说，当两只动物有合作的动机时，它们必须很有说服力地向对方表达善意。为了打消对方的疑虑，它们向对方表达友好时必须附上自己的代价，使得自己背叛对方时不得不付出昂贵的代价。换句话说，表达方式本身必须是对自己不利的。

定义可能很拗口，但是这是在历史上经常发生的事：在中国历史上，国家和国家之间签订盟约，为了表示自己对盟约的诚意，经常会互质。即互相送一个儿子（有些时候甚至会送太子，即皇位继承人）去对方国家做人质。在这种情况下，为取得信任而付出的代价就是君主和儿子的亲情，以及十几年的养育。

比特币的工作量证明很好地利用了不利原理解决了一个自己网络里的社会问题：产生一个新区块是建立在耗时耗力的巨大代价上的，所以当新区块诞生后，某个旷工要么忽视它，继续自己的新区块寻找，要么接受它，在该区块之后继续自己的区块的挖矿。显然前者是不明智的，因为在比特币网络里，以最长链为合法链，这个矿工选择忽视而另起炉灶，就不得不说服足够多的矿工沿着他的路线走。相反要是他选择接受，不仅不会付出额外的辛苦，而且照样可以继续自己的更新区块的挖矿，不会再出现你走你的我走我的，是一个全网良性建设。比特币通过不利原理约束了节点行为，十分伟大，因为这种哲学可以用到如今互联网建设的好多方面，比如防垃圾邮件、防 DDoS 攻击。

PoW 共识协议的优点是**完全去中心化，节点自由进出**。但是依赖机器进行数学运算来获取记账权，资源的消耗相比其他共识机制高，可监管性弱，同时每次达成共识需要全网共同参与运算，性能效率比较低，容错性方面允许全网 50% 节点出错。

- 目前比特币已经吸引全球大部分的算力，其他再用 PoW 共识机制的区块链应用很难

获得相同的算力来保障自身的安全。

- ❑ 挖矿造成大量的资源浪费。
- ❑ 共识达成的周期较长。

4.6 股权权益证明——PoS

股权权益证明（Proof of Stack，以下简称 PoS）现在已经有了很多变种。最基本的概念就是选择生成新的区块的机会应和股权的大小成比例。股权可以是投入的资金，也可以是预先投入的其他资源。

PoS 算法是针对 PoW 算法的缺点的改进。PoS 由 Quantum Mechanic 2011 年在 bitcointalk 首先提出，后经 Peercoin 和 NXT 以不同思路实现。PoS 不像 PoW 那样，无论什么人，买了矿机，下载了软件，就可以参与。PoS 要求参与者预先放一些代币（利益）在区块链上，类似将财产存储在银行，这种模式会根据你持有数字货币的量和时间，分配给你相应的利息。用户只有将一些利益放进链里，相当于押金，用户才会更关注，做出的决定才会更理性。同时也可以引入奖惩机制，使节点的运行更可控，同时更好地防止攻击。

PoS 运作的机制大致如下。

1）加入 PoS 机制的都是持币人，成为验证者（validator）;

2）PoS 算法在这些验证者里挑一个给予权利生成新的区块。挑选顺序依据持币的多少;

3）如果在一定时间内，没有生成区块，PoS 则挑选下一个验证者，给予生成新区块的权利;

4）以此类推，以区块链中最长的链为准。

PoS 和 PoW 有一个很大的区别：在 PoS 机制下，持币是有利息的。众所周知，比特币是有数量限定的。由于有比特币丢失问题，整体上来说，比特币是减少的，也就是说比特币是一个通缩的系统。在 PoS 模式下，引入了币龄的概念，每个币每天产生 1 币龄。比如你持有 100 个币，总共持有了 10 天，那么，此时你的币龄就为 1000，这个时候，如果你发现了一个 PoS 区块，你的币龄就会被清空为 0。你每被清空 365 币龄，你将会从区块中获得一定的利息。因此，PoS 机制下不会产生通缩的情况。

和 PoW 相比，PoS 不需要为了生成新区块而大量的消耗电力，也一定程度上缩短了共识达成的时间。但缺点是：**PoS 还是需要挖矿**。

4.7 委托权益人证明机制——DPoS

委托权益人证明机制（Delegated Proof of Stake，以下简称 DPoS）机制是 PoS 算法的改进。笔者试着以通俗易懂的方式来说明这个算法。

假设以下的场景：百花村旁有一座山叫区块链山，属村民集体所有。村外的 A 公司准

备开发区块链山的旅游资源。A 公司和村民委员会联合成立了百花旅游开发有限公司，签了股份制合作协议。以下是春节假期期间发生在村民李大和柳五之间的对话：

李大：关于旅游开发区块链山，村民委员会和 A 公司签约了。

柳五：那我们有什么好处？

李大：我们都是区块链旅游有限公司的股东了。

由于村民都是股东，所有村民就是区块链山的权益所有人。

柳五：股东要干什么工作呢？

李大：关于区块链的开发的重要决定，股东都要投票的。

柳五：那可不成。春节后我要出去打工，在哪儿还不一定呢。哪有时间回来投票。

李大：不要紧，我们可以推选几个代表，比如王老师，他会一直留在村办小学教书，不会走的，而且人又可靠，讲信用。

柳五：我也推选王老师，代表我们在重大决议上投票。

王老师在这里就是委托权益人（也叫见证人）。DPoS 算法中使用见证人机制（witness）解决中心化问题。总共有 *N* 个见证人对区块进行签名。DPoS 消除了交易需要等待一定数量区块被非信任节点验证的时间消耗。通过减少确认的要求，DPoS 算法大大提高了交易的速度。通过信任少量的诚信节点，可以去除区块签名过程中不必要的步骤。DPoS 的区块可以比 PoW 或者 PoS 容纳更多的交易数量，从而使加密数字货币的交易速度接近像 Visa 和 Mastercard 这样的中心化清算系统。

李大：我们集体推举王老师的人，每年给王老师一点补偿，因为代表我们参加 A 公司的董事会也很花时间，挺累人的。

柳五：成啊！

权益所有人为了见证人尽量长时间在线，要付给见证人一定的报酬。

柳五：我还准备推荐陶大妈。文化高，人也好，也会一直留在村里。

李大：陶大妈身体不好，还是不要干这个差事了。

见证人必须保证尽量在线。如果见证人错过了签署区块链，就要被踢出董事会。不能担任见证人的工作。

村民选举出几个见证人后……

柳五：这次怎么选出了赖大这家伙。这家伙一贯不干好事。我退出！

如果权益所有人不喜欢选出来的见证人，可以选择卖出权益退场。

DPoS 使得区块链网络保留了一些中心化系统的关键优势，同时又能保证一定的去中心化。见证人机制使得交易只用等待少量诚信节点（见证人）的响应，而不必等待其他非信任节点的响应。见证人机制有以下特点。

- 见证人的数量由权益所有者确定，至少需要确保 11 个见证人。
- 见证人必须尽量长时间在线，以便做出响应。
- 见证人代表权益所有人签署和广播新的区块链。

- 见证人如果无法签署区块链，就将失去资格，也将失去这一部分的收入。
- 见证人无法签署无效的交易，因为交易需要所有见证人都确认。

4.8 共识算法的社会学探讨

对于分布式系统的拜占庭问题，从计算机科学的角度，FLP与CAP定理已经告诉我们无解。研究人员及科学家只有从其他地方寻找灵感。其实并不用花太多时间，他们就会发现，真实的人类世界就是一个分布式系统。如果科技畅销书《三体》的世界真的存在，那么太阳系和三体人所在半人马座的星球同时发生了爆炸，对于我们地球人而言，肯定是太阳系的爆炸先发生，因为光肯定是先到达地球。而在三体人看来，他们会首先观测到半人马座的爆炸。对于同样的事件，不同的系统接收到事件的顺序是不一样的。不同的系统运行速度也是不一样的。再加上通信的信道是有问题的。在上面三体人的例子里，我们假设光线的传递是毫无障碍的。但是如果光线被传播途中的黑洞给吞噬了，消息永远接收不到怎么办？

比特币的天才之处在于参照人类社会的组织方式和运作方式，引入了共识机制。一个交易的成立与否，也就是分布式账本的记账权，经由特定共识机制达成的共识来决定。共识，是一个典型的社会学概念。本章中描述的各种共识算法，读者应该都有似曾相识的感觉。

PoW，我们可以叫它"范进中举"。范进用了大半辈子学习一种无用的八股文写作，如同比特币矿工用算力来算题，关键是算的题毫无意义。有朝一日，运气好，就可以有权打包所有他认可的交易。

PoS是用户要预先放入一些利益，这是不是很像我们现实世界中的股份制。人们把真金白银兑换成股份，开始创业。谁的股份多，谁的话语权就大。

DPoS机制，特别像我们的董事会。选举出代表，代表股东的利益。被选出的代表，一般来说，成熟老练、阅历丰富。不但能快速地处理日常事务，同时也能很好地保护股东的利益。

Paxos、Raft、PBFT则很像我们生活中的操练队列，通过互相间的消息、口令来达成一致。每排的排头作为Leader，而每排的其余人都以排头为目标，调整自己的行动。瑞波共识算法，初始状态中有一个特殊节点列表，就像一个俱乐部，要接纳一个新成员，必须由51%的该俱乐部会员投票通过。共识由核心成员的51%权力投票决定，外部人员则没有影响力。由于该俱乐部由"中心化"开始，它将一直是"中心化的"，而如果它开始腐化，股东们什么也做不了。与比特币及点点币一样，瑞波系统将股东们与其投票权隔开，因此比其他系统更中心化。

如果我们去看Lamport关于分布式系统共识的论文，就会发现论文是以议员、法案和信使作为阐述理论的样例，读起来不太像一篇计算机论文。

在此可以做一个总结了。传统的、纯正的计算机算法对分布式系统的拜占庭问题已经无处着力了（参考FLP与CAP定理）。所以在分布式系统的研究中引入了一些社会学的理论和概念，包括上述的博弈论，生物学原理，等等。我们可以把每一个计算机节点想象成

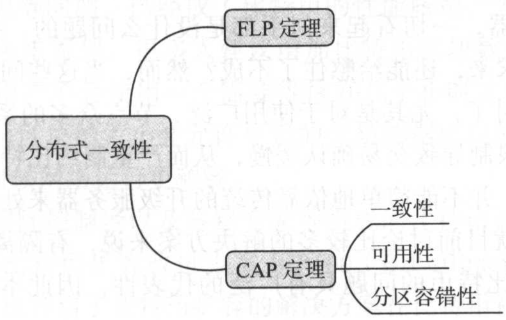
一个单元。而计算机网络就是一个个单元组成的社会，我们该如何给这个计算机节点组成的社会设计规则呢，以保证：

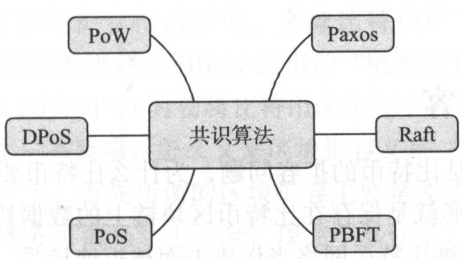
- 少量节点太慢，或者故障崩溃的情况下，整个网络还能输出正确的结果；
- 整个网络的响应不能太慢。买一杯咖啡要等一小时是不可接受的；
- 计算机网络出现分区（网络上的某些节点和其余节点完全断开）的时候，仍然能够稳定输出正确的结果；
- 整个系统能够稳定地运行，输出稳定的结果。

我们可以借鉴人类历史上的社会机制、激励机制，达成上述的功能。我们有理由相信，互联网或者分布式网络系统与现实的社会运作有着千丝万缕的联系，正因为如此，区块链的发展并不是冥冥之中的产物。

4.9 知识点导图

本章主要介绍了拜占庭将军问题，以及计算机科学家对拜占庭将军问题得出的研究结果：FLP、CAP 定理。区块链技术创造性地提出了各种各样的共识算法，尝试解决拜占庭将军问题。本章对主流的各种共识算法也尽力做了通俗易懂的描述。





Chapter 5 第 5 章

区块链扩展：扩容、侧链和闪电网络

天下武功唯快不破，因此在互联网时代，很多软件采取的都是先开发一个简化版，然后再快速迭代，这边功能少了就加上，那边功能弱了就补上，数据量大了就增加存储器，性能不够了就增加服务器，一切看起来似乎都是没什么问题的，扩展嘛，什么纵向扩展、横向扩展，一切跟着需求来，还能给憋住了不成？然而，当这些问题进入到区块链的场景，情况就有些不那么好对付了，尤其是对于使用广泛、节点众多的系统（如比特币），典型的问题就是由于区块容量限制导致交易确认缓慢，从而严重制约比特币网络的交易处理能力，而比特币是分布式结构，并不能简单地依靠传统的升级服务器来处理。为了解决这些问题，社区发起了各种讨论，就目前讨论比较多的解决方案来说，有隔离见证、直接扩容、侧链以及闪电网络。考虑到比特币的问题具有广泛的代表性，因此本章就以比特币为场景来叙述。

5.1 比特币区块扩容

首先我们要解释什么是比特币的扩容问题，为什么比特币要扩容。比特币的扩容问题来自一个很直接的现实，那就是保存在比特币区块链上的数据块的物理大小限制是 1MB。任何大于 1MB 的区块都会被比特币网络当作攻击而被拒绝接受，这是当初由中本聪对比特币核心的设计决定的，逻辑规则都写在了源码中。

大家都知道比特币其实就是一个分布式的公共记账（数据库）系统。也就是说，比特币本质其实是拿来记账用的，当然大部分情况是对比特币这个数字货币记账。

由比特币底层技术发展起来的区块链技术也是对各种承载价值以及数据状态进行记账。

比特币的数据包含交易数据及其对应的货币台账，我们简单想一想，最主要的问题来自大家在日常交易转账的时候，需要不断地把交易数据发送到网络中的节点，经过矿工打包成区块后广播给其他节点，每个节点验证通过后独立的加入自己本地的区块链账本数据库中。随着时间的推移以及比特币生态系统的扩张发展与深入应用，用户数越来越广泛，交易次数也越来越多，网络中等待确认的交易则排起了长长的队伍，这时我们就会遇到单个区块的容量限制问题了。

而现在这样的容量限制问题已经发生，比特币网络已经由于交易缓慢而变得拥挤不堪。由于区块大小 1MB 的限制，单个区块只能容纳很有限的交易事务，在一个区块的结构中，区块头也就是区块的摘要信息字段占据了 80 个字节，每条交易事务平均在 200 字节左右，往多了算，假设区块中的交易都是一对一的简单交易，单个区块能够容纳的交易数也就 8000 左右，而实际上根据目前的使用统计，单个区块容纳的交易数才 1700 多，就这还得要等上间隔 10 分钟的打包确认，因此算下来，交易速度最高大概 1 秒钟只能处理 3 笔交易[1][2]。要知道已经被市场所广泛熟悉与习惯的支付手段（像 VISA、Master 卡等信用卡银行卡）交易处理速度每秒钟高达几千笔交易。

由于这个区块大小的限制，很多用户为了能够尽快让网络确认自己的交易，不得不增加交易手续费（比特币中矿工节点会按照手续费高低进行优先级处理）。大量交易费用的增加以及交易处理严重延迟等问题，已经成了比特币的性能瓶颈，大大限制了比特币的应用和发展。为了比特币的未来着想，很多人建议增加比特币区块数据的大小。原因很简单，因为大部分商家和最终用户不会使用一个需要等待好几个小时才能确认一笔交易的系统。理论上来讲，增加比特币区块的大小会允许更多的交易数据可以放到一个数据块中，使得更多的人使用比特币的时候网络运行更顺畅。

为此，比特币网络实际控制者以及各路专家等组成的比特币社区提出了很多对比特币扩容的方案。2015 年，比特币扩容改进方案 BIP100（BIP=Bitcoin Imprevement Proposal）和 BIP101 先后被提出，也开启了比特币扩容的解决方案在比特币社区激烈的冲突和争论。

2017 年 7 月 21 日，真正的扩容行动开始，全球比特币矿工开始锁定一个扩容软件升级。这次升级是基于比特币改进建议 BIP91 提出的“隔离见证”（Segregated Witness = SegWit）的方案，并计划在 2017 年 11 月份将比特币区块大小从 1MB 提升到 2MB[3]。

大家在这里可能会产生一些疑惑，怎么又是区块扩容又是“隔离见证”，这里面都是些什么关系呢？我们来解释一下，首先所谓的区块扩容主要是要增加区块中容纳交易事务的区块体的空间大小，这个地方可谓是寸土寸金，现在不够住了，怎么办呢？相信大家根据生活经验也能给出两个一般性的做法：

---

[1] Mike Orcutt (19 May 2015). “Leaderless Bitcoin Struggles to Make Its Most Crucial Decision”. MIT Technology Review. Retrieved 15 November 2016.

[2] http://money.cnn.com/2017/08/01/technology/business/bitcoin-cash-new-currency/index.html.

[3] Hertig, Alyssa (July 21, 2017). “BIP 91 Locks In: What This Means for Bitcoin and Why It’s Not Scaled Yet”.

1）增加区块空间的大小，宽敞又明亮；

2）缩小交易数据的尺寸，节能又环保。

第一种方案显然是最符合人们一般性思维的，这也是社区中坚持区块直接扩容一派的思想，那么第二种的缩小交易数据尺寸是什么意思，这里需要解释一个概念，那就是“隔离见证”，我们来简要说明一下。

“隔离见证”，英文是 Segregated Witness，我们知道在比特币的交易数据结构中，是通过发起者签署自己的 UTXO（未花费交易输出），然后填上接收者的地址而建立起来的，过程类似于签署支票，一张支票就相当于一条比特币的交易事务，签署 UTXO 就相当于支票签名，也就是所谓的“见证”，这是用来确认支票合法性的。我们知道，支票上的关键内容无非就是签名和接收方以及支付金额，那么如何来确定这张支票数据的唯一性或者说完整性呢？在比特币中会对每一条交易事务数据进行一次哈希计算，得到一个事务 ID，在计算这个事务 ID 的过程中，都有哪些数据参与了计算呢？答案是整条交易事务，包括那个签名。那么这里就有可以探讨的余地了，一切就围绕这个签名来展开讨论，我们从比特币的交易历史数据中随便截取某笔交易的签名信息来看一下：

```
    "scriptSig": {
        "asm": "3044022065c13d7cf6557af8ad45dbfd2b0847950e0f11e3c0eb2468ca9a8ad612e2
1d5b022064bea5eb078b7c89aad63730dbde1e8dd7dbaa0614b2c0809fa1baedf66eac21[ALL] 036b071
44610d46dbe4bdcc2ff3ecd68627e645027aac62cc5e9147a6575f7cb55",
        "hex": "473044022065c13d7cf6557af8ad45dbfd2b0847950e0f11e3c0eb2468ca9a8ad612
e21d5b022064bea5eb078b7c89aad63730dbde1e8dd7dbaa0614b2c0809fa1baedf66eac210121036b071
44610d46dbe4bdcc2ff3ecd68627e645027aac62cc5e9147a6575f7cb55"


    }
```

可以看到这个签名信息占据的空间还是不少的，如果能够把这块签名信息从交易事务中隔离开，存储在另外一边，那就能省出一块空间来容纳更多的交易数据。这些签名信息的主要作用就是见证交易数据的来源合法性，而实际上见证的过程只需要进行一次就行了，矿工负责见证交易数据是否得到了合法的授权，其他普通的节点只关心接收的结果，见证过后这些签名数据实际上没多大用处，节点在接收时可以丢弃这部分数据。这种将见证信息与交易数据隔离开的设想也就是“隔离见证”的意思。实际上“隔离见证”还在一定程度上能解决一个叫“交易延展性”的问题。

如上所述，交易事务 ID 在计算时将计算整条含签名的交易数据，而这个签名是可以被更改掉的，因为签名有很多种写法，攻击者无法修改交易事务中的输入和输出，但是却能重新修改签名，从而导致交易事务 ID 的计算值发生变化，一旦被攻击者更改，虽然不能被窃取比特币，但是却有可能导致交易不被网络确认（网络中会同时存在没有被修改过和被修改过的交易事务，这会导致冲突），而隔离开签名信息后，交易事务一旦发起将会完全固化。由于“隔离见证”的这些特点，因此这种方案也有不少人支持。

至此，比特币的扩容方案就有了如下的选择：

1）进行“隔离见证”并扩容区块；

2）仅进行隔离见证，区块容量保持不变；

3）仅扩容区块，不进行隔离见证。

这些方案各有不同的社区成员支持，这些成员主要包括比特币核心客户端维护团队、各大矿池以及比较有影响力的开发团队和广大的社区用户，对于到底选择何种方案，各方进行了旷日持久的争论。我们不去细究这里面潜在的各方利益问题，单就技术角度而言，有一个问题是确定的，那就是无论选择何种方案，都避免不了会产生比特币主链的分叉，“隔离见证”或者扩容，都需要修改现有的比特币源代码。这对于传统软件来说是分分钟的事，无论怎么升级，只要保持兼容原有的数据格式就行了，可是对于比特币这种区块链应用程序，首先它是分布式的，谁也没有能力强制大家共同升级到一个新版，那就势必会导致一旦新版本发布后，网络中会同时存在老版本和旧版本的节点，而对于矿工或者说矿池而言，也会选择不同的支持方案，那么网络中新打包出来的区块有些是旧版本格式的，有些是新版本格式的，彼此之间无论如何也很难做到完全一致，这样就会导致原先单一的主链由于后续产生了不同格式的区块而分叉出两条链，甚至多条链，非但如此，当某一方的挖矿算力明显占据优势的时候，相对弱的那一方产生的区块链甚至会因为得不到大多数的节点背书而沦为孤儿链，这会使得原本牢固的去中心化区块链共识网络变得脆弱，这不是我们所愿意见到的。当然，从长远来看，如果解决一个问题不得不付出一些代价，分叉也并非完全不能接受，只是这个过程如何过渡好需要仔细衡量。

我们来看一下这些年为了比特币区块扩容发生的那些事。

- 2015-Bitcoin XT（比特币扩展），2015年提出通过增加数据块的大小限制来提高交易处理效率，最早建议数据块大小是8MB，然后数据块大小根据交易数据情况自动增长，每两年大小翻一倍等，但是事后这个建议没有得到足够的支持而最后未被接受。
- 2016-Bitcoin Classic（比特币经典），2016年也提出通过增加数据块的大小限制来提高交易处理效率，但是没有Bitcoin XT那么激进，最早提出区块大小从1MB扩容到2MB，然后在后期决定把区块大小上限交给矿池和交易节点来决定，不过并没有得到比特币核心开发团队的支持。
- 2016-香港共识，2016年2月21日，在香港数码港，由比特币业界代表和开发社区代表参与的圆桌会议达成了扩容共识：软件激活由比特币核心开发人员执行在2015年12月提出的隔离见证，并将区块大小限制扩充到2MB；很遗憾，此次共识达成的两个行动都逾期了。
- 2016-Bitcoin Unlimited（BU-比特币无极限），在Bitcoin XT和Bitcoin Classic扩容方案夭折之后，Bitcoin Unlimited提出增加区块大小的方案是完全取消区块大小限制，让用户通过查看大多数共识区块的大小决定并自行设置自己区块的大小。这个方案得到了不少矿池的支持。2017年1月发布1.0.0版本得到了包括Antpool、bitcoin.

com、BTC.TOP、GBMiners 和 ViaBTC 等矿池的支持。至 2017 年 3 月，全球大概由 11% 的节点运行 BU 升级版。但是，比特币的扩容并不是简简单单地将区块大小限制取消就万事大吉，扩容涉及很多方面的技术细节并需要大量的测试。果然，2017 年 1 月 19 日 BU 发现重大漏洞，由 bitcoin.com 矿池打包出第 450 529 无效区块，1 月 31 日打补丁，2017 年 3 月 14 日，BU 全节点遭到攻击，BU 节点数量大量宕机，然后代码漏洞一个接一个，2017 年 4 月 24 日，70% 的 BU 节点因为内存泄露而出现系统崩溃[㊀]。

- 2017-BIP148，一个通过用户激活的软分叉比特币扩容方案被提出。BIP148 打算绕过矿工和矿池的支持，在 2017 年 8 月 1 号启动一个用户欢迎程度来激活（UASF）的比特币扩容软件升级方案，该方案建议 2017 年 8 月 1 号起，激活比特币的隔离见证（SegWit）功能。
- 2017- 纽约共识（SegWit2x），2017 年 5 月，“数字货币集团”公布一个扩容方案，也就是 SegWit2x，即著名的“纽约共识”：先在获得 80% 的比特币算力支持基础上首先激活隔离见证方案，并在 6 个月后获得 80% 比特币算力支持的时候激活将区块大小从 1 MB 扩展到 2MB 的升级。时至 2017 年 7 月中旬，矿工和矿池基本一致同意在 2017 年 8 月 1 号前实施激活隔离见证（Segwit2x）方案[㊁]。
- 2017-Bitcoin Cash（BCC），BCC 是 2017 年 8 月 1 日比特币硬分叉产生的一个新的比特币区块链变种。当比特币矿池和交易所 ViaBTC 为了对抗隔离见证（SegWit），挖出第 478 559 区块，正式宣告比特币历史上的第一次硬分叉[㊂]。比特币硬分叉后产生两个新的币种：比特币(BTC) 和比特币现金（BitCoin Cash，以下简称 BCC），硬分叉前的比特币所有者会自动分配同时拥有分叉后的比特币（BCC）和比特币现金（BCC）。BCC 的区块大小从 1MB 扩容到 8MB，而不引入隔离见证。在 2017 年 8 月 1 日午夜之后，BCC 的市场市值达到继比特币和以太坊之后的第三大市值，随后越来越多的交易所也慢慢开始支持 BCC[㊃]。

可以看到，为了一个区块扩容竟然产生了这么多的讨论和争议，一个初看起来似乎很简单的问题却包含了各种技术考量，不过就在比特币社区在方案上悬而未决的时候，比特币的兄弟莱特币在 2017 年 5 月却率先完成了隔离见证，莱特币是通过对比特币源码的简单修改而来的，因此在血统上很接近比特币，莱特币成功实行隔离见证激活的经验也给比特币社区做了一个示范和参考，就技术方案而言，莱特币具体是通过一个叫“用户激活软分叉”的方案来进行的，我们来了解一下。

㊀ Quentson, Andrew (24 April 2017). “Bitcoin Unlimited Nodes Crash Due to Memory Leaks”. Cryptocoinsnews. Retrieved 15 March 2017.

㊁ CNBC (July 14, 2017). “Dispute could mean financial panic in bitcoin”. Associated Press.

㊂ Coleman, Lester (July 25, 2017). “Bitmain Clarifies Its ‘Bitcoin Cash’ Fork Position”. CryptoCoinsNews.

㊃ CNorrie, Adam (July 29, 2017). “Bitcoin Cash: Another Fork in the Road for Bitcoin”. CryptoCoinsNews.

用户激活软分叉（User-Activated Soft Folk，UASF）是一个很有意思，也备受争议的软分叉升级模式。主要是为了避开掌握着大量算力的矿工和矿池的反对，而将支持升级的决定权交给矿工和矿池之外的所有节点和用户。这样就使得区块链核心研发团队可以避免等待掌握大量算力的矿池节点的支持。将软件升级支持设置在运行全节点的交易所、钱包，还有莱特币使用者手中。因为只要是区块链全节点（full node），都具有校验区块和交易数据合法性的功能。当交易所征集到大部分用户的签名和支持后，新的升级版软件才会被事先已经安装的软件激活。这样所有支持软分叉的交易所和用户都会安装新版规则和共识的软件，从而成功实现大多数人支持的软分叉。

不过这样聪明的软分叉方式有一个问题，就是开发成本太高，软件更新周期太长，没有在掌握算力的矿池那里直接升级来得高效、直接、快速。当然，这种明目张胆绕开掌握大量算力的矿池的做法也会引起不可预见的后果，那就是其矿池节点也可以自行选择修改规则发布自己的升级版软件而强制区块链硬分叉。

我们可以看到，对于区块链这种新型的网络软件结构，有其明显的优势，但是也有明显的问题所在，就区块扩容这个问题而言是具有代表性的，比特币、莱特币、以太坊等其实都会有这样的问题，区块容量爆炸一直都是这个领域的难点问题，尤其是对于使用广泛的区块链系统，这个问题的严重程度尤甚。以太坊相对比特币、莱特币等支持了更复杂的智能合约，并且使用广泛，问题也就更多，目前以太坊社区提出的解决方案有提高 Gas 限制以及分片，提高 Gas 限制相当于提高用户的使用成本，与其说这是一种技术方案，不如说是一种经济制裁方案，分片的意思是将区块数据按照某种分类存储在不同的节点上，而不像现在所有的节点都保存同样的副本数据，不过这种方案的争议也是很大的，可靠性和安全性都有待验证。说到这里，有读者可能会提出，既然靠单个链内很难完善地解决这个问题，那有没有可能将某些交易事务移出去呢？答案是可行，链内方案的地雷太多，那么看看链外的方案如何，接下来介绍的侧链、闪电网络以及多链就是这种思路。

5.2　侧链技术

在了解侧链技术之前，我们先看如下的对话。

Alice：我有两个不同的数字货币钱包：比特币和以太币，我可以将比特币从比特币钱包地址转到以太币钱包地址吗？

Bob：一般情况下当然不可以啦，比特币和以太币是两个完全不同技术和构架的区块链，它们的价值不能直接转换。

Alice：那有什么办法可以做到两个不同的区块链数字货币之间直接做价值转换？

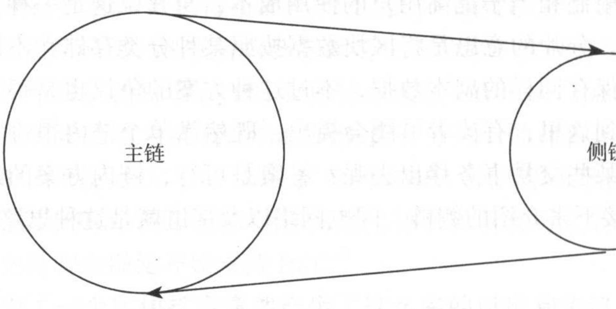
Bob：那就必须引入侧链，侧链协议可以将比特币从主链上转移到侧链上来。但是需要在比特币主链上先冻结，然后在别的链上激活。

我们知道区块链本质是公共账本技术，主链承载的都是账本核心交易数据（或价值）。

当一笔交易的信息太大或复杂的时候，会在不影响账本数据一致性和安全性的基础上通过引入侧链的技术来分流数据量（或价值）。

传统意义上的侧链就是指将比特币（价值）从比特币主链上来回转移到与比特币完全不同特征和技术构架的区块链上。所以侧链不是指比特币（区块链）主链上的某个部分，而是指遵循侧链协议的所有区块链，侧链这个名词是相对于比特币主链而言的。侧链协议是指可以让比特币和其他区块链账本资产在多个区块链之间来回转移的协议。大家需要注意的是，主侧是相对的，没有说哪种链必须是主链或者是侧链，根据需要，任何一种链都可以成为另外一种链的侧链或者是主链，比如比特币可以成为莱特币的侧链，以太坊可以成为比特币的侧链等，侧链可以是完全独立的链，也可以是必须依赖主链生存的链。

所以，只要实现侧链协议，现有所有的区块链、比特币、以太坊、比特币现金、莱特币、瑞波币等彼此竞争的区块链都可以成为侧链，不过，目前侧链的实现还是主要来自比特币的各种侧链系统，把比特币的资产从比特币主链上转移下来，这开辟了一条通道，让用户可以通过已经拥有的比特币资产，去培养和孵化一些更创新、更适用的数字货币系统或者其他更丰富的应用，由于比特币本身已经是目前使用最广泛的区块链系统，因此通过侧链的扩展，可以充分发挥比特币网络的价值和作用，比较著名的比特币侧链有 ConsenSys 的 BTC Relay、Rootstock 和 BlockStream 推出的元素链，非比特币的侧链如 Lisk 和国内的 Asch。我们看下主链和侧链的关系：



如图所示，站在软件的角度，其实就是两种不同的软件进行数据交互，一方以另一方的功能和数据作为依托来开展其他的业务功能。如果将图中的侧链换成一个普通的软件客户端（如钱包软件），那就不能叫侧链了，因为钱包不是一个区块链系统，这样讲是为了让大家能够比较容易地理解侧链的角色作用。接下来我们以比特币为例看一下侧链的工作方式。

（1）单一托管

为了将比特币从主链上移动到侧链，比特币区块链上的比特币必须首先在主链上被冻结，然后在侧链上激活，这叫双向锚定。最简单的实现双向锚定的侧链就是将比特币主链上的资产发送到一个单一托管方，并在侧链上激活。其实，这样单一托管的方式，由一个机构去主链上冻结资产的侧链跟一家现实中的数字资产交易所的方式都很类似，所以这样

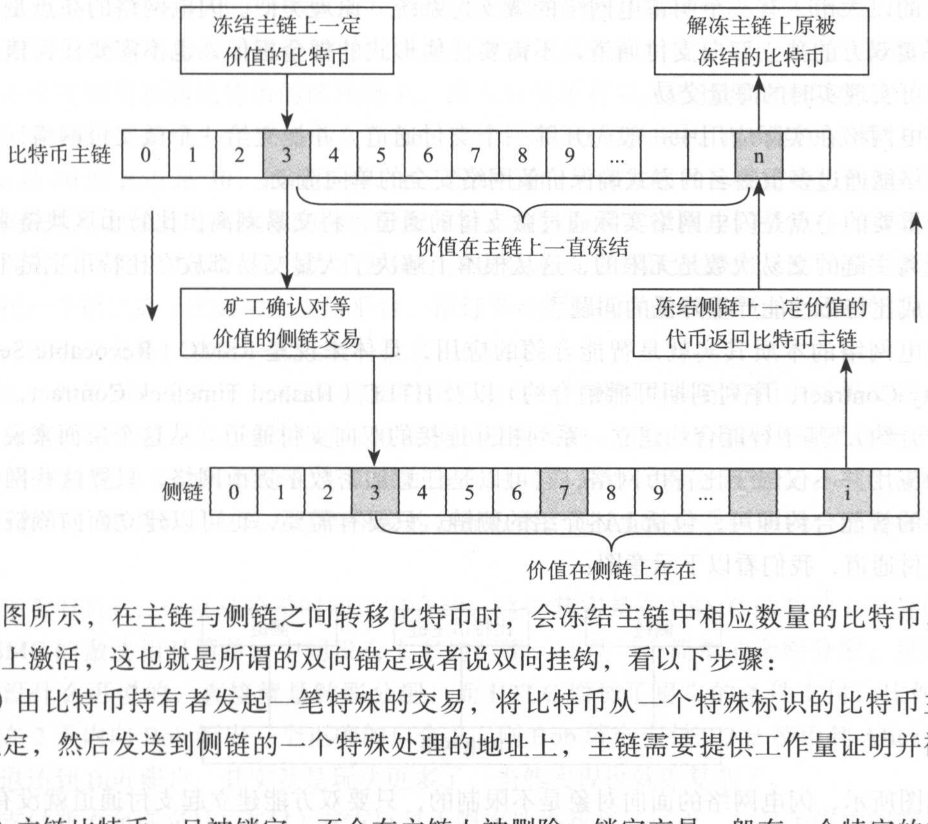
最明显的问题就是：这是完完全全的中心化的解决方案[⊖]。

我们平时常用的比特币钱包也是一种单一托管模式的侧链技术。它保证你的资产冻结在一个节点上保管或者应用。

（2）合约联盟

简单地说，就是比特币主链上冻结的资产通过一个多重签名的地址控制，这个类似于一份智能合约，双方或者多方约定一个公证保管规则。比起第一种单一托管，这种方式更加增强了安全性，也使得侧链协议实现得更加顺畅。

除了以上两种方式，还有很多种技术可以实现将区块链主链上的资产发送到目标侧链上，或者从目标侧链发送到主链，为了更好地理解，我们看一下侧链双向锚定的思路和步骤，先来看一幅示意图：



如图所示，在主链与侧链之间转移比特币时，会冻结主链中相应数量的比特币，然后在侧链上激活，这也就是所谓的双向锚定或者说双向挂钩，看以下步骤：

1）由比特币持有者发起一笔特殊的交易，将比特币从一个特殊标识的比特币主链地址上锁定，然后发送到侧链的一个特殊处理的地址上，主链需要提供工作量证明并被侧链认可；

2）主链比特币一旦被锁定，不会在主链上被删除。锁定交易一般有一个特定的等待确认期，等足够大量随机的节点确认，更有效地防止被假冒和攻击；

3）由于侧链已经同意作为比特币的侧链，侧链将产生跟主链转移过来的资产对等的侧链资产，并设置合适的所有权，完全按照侧链的游戏规则进行；

⊖ Enabling Blockchain Innovations with Pegged Sidechains. https://www.blockstream.com/sidechains.pdf.

4）上述逻辑一般是对等的，可以将资产从比特币主链上转移出来，也可以用同样的道理将资产转移回来。

通过建立侧链，在保证比特币价值的基础上把交易 / 资产转移到别的完全不同构架、技术和共识机制的新区块链上，也可以说是解决比特币扩容和性能瓶颈的最好方案。很多比特币改进建议，都是各种侧链的变化。

5.3 闪电网络的设计

闪电网络（Lightning Network）是一个点对点对等网络，完全去中心化的数字货币微支付系统。这个微支付系统的理念适用于比特币、以太币和莱特币这样的数字货币，针对以太坊上的以太币，有一个叫雷电网络的微支付系统，原理类似。闪电网络的亮点是它完全基于买卖双方的独立双向支付通道，不需要任何形式的押金担保，也不需要任何信任的第三方即可实现实时的海量交易。

闪电网络在实际应用中一般先开辟一个支付通道，并提交给一个微支付网络，这个微支付网络能通过多重签名的方式确保价值网络安全的单向流动。

最重要的一点是闪电网络实际通过微支付的通道，将交易剥离出比特币区块链来进行，而且剥离主链的交易次数是无限的，这从根本上解决了大量交易都放在比特币主链上进行，从而造成比特币性能严重降低的问题[⊖]。

闪电网络的本质其实就是智能合约的应用，具体来说是 RSMC（Revocable Sequence Maturity Contract，序列到期可撤销合约）以及 HTLC（Hashed Timelock Contract，哈希时间锁定合约），基于智能合约建立一系列相互连接的双向支付通道。从这个层面来说，闪电网络的应用并不仅限于比特币网络，它可以是任意加密数字货币网络，只要这些网络能支持需要的智能合约即可，包括上述介绍的侧链。只要有需要，也可以建立面向侧链的闪电网络支付通道，我们看以下示意图：



如图所示，闪电网络的面向对象是不限制的，只要双方能建立起支付通道就没有问题，由于需要智能合约的支持，因此对于比特币这种自定义脚本编程能力有限的系统，需要增加一些必要的操作指令，同时由于闪电网络在配合使用的时候，需要经常打开和关闭支付通道，这会加剧原本就拥堵的比特币网络，因此如果比特币系统能够良好地实现隔离见证

⊖ The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf.

或扩容，那对于闪电网络的真正落地使用能起到很好的促进作用。大家在查阅一些资料的时候，会经常看到闪电网络与隔离见证的字眼，需要注意的是，隔离见证并不是闪电网络实现的必要条件，只不过一定程度上可以简化闪电网络的设计。

我们现在解释一下 RSMC 的机制。“序列到期可撤销合约”，这个名词初看起来，很难明白是什么意思，我们来举个例子，闪电网络是通过支付通道来进行收支业务的，在通道创建的初期会记录一个初始的资金分配方案，这个资金从哪来呢？比如 Alice 与 Bob 之间由于双方的业务关系需要长期频繁转账，而且每次转账也都是小额，为了方便，他们打算平时先不进行实际的转账，而是先记个账，到一个时间后再来算个总账。于是他们共同拿出一笔钱开设一个基金账户，假设 Alice 和 Bob 都拿出 50，则初始的分配方案就是 Alice 为 50，Bob 也为 50，这个通道的设立会记录在比特币区块链上。随着业务的发展，分配开始发生变化了，Alice 支付了 10 给 Bob，此时最新的分配方案就变成了 Alice 是 40 而 Bob 是 60，双方共同签名作废了之前的分配方案，更新了最新的余额分配，不过这份新的分配方案并不会立即更新到比特币的区块链上，因为后续还有双方的日常业务发生，因此只是记录在闪电网络区块链上，果然不久，Bob 又支付了 30 给 Alice，此时新的分配方案变成了 Alice 是 70 而 Bob 是 30，依此类推，在一段时间内，双方都只是在比特币的链下（闪电网络中）频繁地记录着每一次新的余额分配方案。这种方式其实跟我们平常在一家饭店订餐或者订购鲜花等都很类似，我们为了方便往往也会先预交一部分基金，为了信用保障，可能会委托一个第三方（比如某个支付平台、预订平台等）托管这个基金，一段时间后，大家签字认可发生的交易，然后一次性做一个真正的结算。

那么，如果到一个点上，Alice 需要用钱了怎么办？她可以向比特币主链提交目前最新的分配方案要求结算，在一段时间内如果 Bob 没有反对，则比特币区块链就会终止通道，并且按照合约规则自动转账分配。如果在这个时间内 Bob 反对并且提交了一个证明，表明 Alice 作弊，使用了一个双方已经作废的分配方案，则 Alice 会受到惩罚，资金将会罚没给 Bob。

再来看 HTLC，也就是哈希时间锁定合约，这个其实是在 RSMC 的基础上更复杂了一层，RSMC 的做法相对简单，中间没有太多的逻辑，就是一个简单的余额分配，只要满足条件就没什么可说的，直接就是转账分配，而 HTLC 增加了更多的条件支付，比如 Alice 如果能在 2 天内向 Bob 给出一个正确的口令 R，则 Bob 就会支付 0.2 比特币到 Alice，逾期则自动退还到 Bob 账户。其实就是玩法更多了，当然实现也就更复杂了。

微支付通道允许交易（支付）双方反复无限次地更新交易过程，并且不将中间交易数据写到公有链上，而是将最后的结果上链，这样允许交易对手双方不需要建立信任关系，降低交易对手风险。中间的交易流程走的也还是真实的比特币，各种换手交易和中间结果不真正上链。

一般情况下，交易过程是指交易双方的余额表从交易前状态更新为交易后状态。最核心的问题就是双方对交易后状态的共同确认。一旦有交易一方反悔或不认账，交易后双方

款项的余额是处于不确认状态的。微支付通道通过建立一个基于时间序列的类似多签名智能合约的交易方式来解决这个彼此不信任的问题：

1）Alice 和 Bob 同意建立一笔交易，但是暂时不在链上公告广播；

2）双方把币打到一个地址上，并提供双重签名，同时一致同意交易前的余额状态并上链；

3）双方同时也建立一笔退款交易，各自拿回自己的币，同时这个退款交易也不上链，这样双方事后都可以修改余额状态；

4）当真正发生交易需要更新余额表的时候，双方都生成一个需要提交更新的余额状态表；

5）这样微支付通道里的双方交易余额表无论在谁手上，都只能有两种状态，维系旧的余额状态或承认新的余额状态；

6）任何一方反悔或者不承认新的交易状态，对手方可以提交证据证明，并通过罚没机制拿走双方共同签名的所有的币；

7）交易一方通过提走交易之前共同签名提供的币，来惩罚反悔或不承认的一方以确保新的交易余额状态得到认可；

8）双方都没有争议之后，在等待一段时间并获得网络认可之后的余额状态上链存证，交易完成。

下面让我们来看具体的交易例子和过程以充分理解闪电网络：Alice 需要通过闪电网络给 Bob 和别的交易对手支付比特币资产。

步骤 1：建立微支付交易通道（双向）

双方同意共同创建一个微支付通道（Micropayment Channel），并往里面放一部分订金，我们假设 Alice 打算给 Bob 支付 5 个比特币，而且 Alice 还想通过这个支付通道经常给 Bob 支付比特币。这样双方协商创建一个彼此对等的单向微支付通道，从而构成一个双路微支付通道（就是说，Bob 对比 Alice 做的事情，自己参照对应反向也做一个同样的动作来建立另一个单向通道）。

为了建立这个通道，Alice 和 Bob 分别往一个 2/2 双人签名的地址发送 5 个比特币，我们暂时叫这个账户"订金交易"，未来所有的后续交易只能从这个"订金交易"里支付。两个人都必须共同签名，同时都生成一对自己掌握的针对这个"订金交易"地址的密码和私钥，各自保留自己的密码，但是将自己的私钥交给对方，算是各签一半。这个时候的余额状态是：

- Address #1: 0-Alice&Bob（5 BTC）；1-Bob（5 BTC）
- Address #2: 0-Alice（5 BTC）；1-Alice&Bob（5 BTC）

Alice 现在需要花钱，然后马上就在"订金交易"的基础上创建一笔交易，这个交易我们暂时叫"承诺交易"，在这笔"承诺交易"中，Alice 把 4 个比特币划给自己，另外 6 个比特币划给 Bob，然后这个交易发到新的两人签名的地址 #3。Alice 发送"承诺交易"的地址 #3 有点"诡异"，那就是 Bob 可以自己独自解锁拿走无论谁确认都是属于自己的 6 BTC，但是前提条件是必须等待当前交易所在区块链区块之后的第 1000 个区块被开挖出来，因为

这个区块被加上了一把“时间锁”。而对于 Alice 而言，她也可以独自打开地址 #3 的锁，条件是 Bob 必须将自己的地址 #3 的密码和私钥都交给 Alice 才行。（由于这个时候 Alice 拿不到 Bob 的密码，所以无法动用地址 #3 的资金，哪怕是她共同签名的 4 BTC。）

Alice 对“承诺交易”签名，但是她没有广播出去，而是将签名后的交易交给 Bob 。与此同时，Bob 也在做同样的事情，在地址 #4 创建自己的“承诺交易”并签名交给 Alice 不做广播。这个时候的余额状态是：

- ❑ Address #3: 0-Alice&Bob（4 BTC）；1-Bob（6 BTC）
- ❑ Address #4: 0-Alice（4 BTC）；1-Alice&Bob（6 BTC）

在交换完所有的“承诺交易”以及各自的私钥之后，各自签名并将自己创建的“承诺交易”广播并确保交易被广播到区块链上，至此双通道微支付通道正式打开。由于“承诺交易”带有时间锁，当正常提交的“承诺交易”经过自己提交地址之后的第 1000 个区块被开挖出来之后。交易由闪电网络确认，最终 #3 的交易结果是：0-Alice（4 BTC）；1-Bob（6 BTC）。#4 的交易结果是：0-Alice（4 BTC）；1-Bob（6 BTC）。

而这个时候任何一方都可以将对手私下交给自己已经“一半签名”的交易进行签名，并由自己广播出去，这样两个“承诺交易”可能发生的情况是：

- ❑ #3：如果 Bob 拿到后提供自己签名并广播出去，需要等 1000 个区块链才能开锁拿到 6 BTC；
- ❑ #4 ：如果 Alice 拿到后提供自己签名并广播出去，也需要等 1000 个区块链才能开锁拿到 4 BTC。

以上一切正常，然后我们考虑一种情况，就是当 Bob 想再支付 Alice 一个比特币的时候，双方都想在原来的微支付通道上更新交易状态，使得交易双方的状态达到 5-5 分成。然后双方做如下几步来达到：

1）双方都再次创建“承诺交易”，分别是 #5 和 #6 ，将 5 BTC 签名分配给自己，然后另外 5 BTC 签名作为 2-2 多重签名的一部分加上时间锁。双方产生新的密码和私钥对，并保管好自己的密码，然后完成自己的一半签名并同私钥交给对方。

2）Alice 和 Bob 都要求必须将第一个“承诺交易”中产生的原来私藏的密码交给对方。

在这个时候，双方都可以将彼此签了一半的新“承诺交易”签名并提交确认。任何签字广播一方的对手都可以立即得到属于自己的那一半比特币。而签字广播人则等 1000 个区块挖出后得到自己的一半，这样，微支付通道的新的状态得到更新。

但是，我们不能防止有人会作恶，那就是：例如 Bob 是否可以考虑侥幸想拿自己可控的 #4 的交易状态再签名广播出去，这样他拿到的将是最初的“承诺交易”#4 里的 6 BTC，以此获利。

其实，现实情况是 Bob 并不能一次获利，因为他的第一个状态签名密码这个时候已经交到 Alice 手里。这时如果 Bob 把 #4 拿出来签名再合法广播出去，Alice 首先马上获得应得的 4 BTC，Bob 自己则需要等 1000 个区块链后才能申请得到 6 BTC。可是 Bob 如果想这

样欺诈是有风险和问题的，那就是这个时候 Alice 已经拿到 Bob 自己的密码与私钥，任何时候都可以开锁获得本来应该属于 Bob 的 6 BTC，这样 Bob 就会偷鸡不成蚀把米。同样，Bob 也拥有 Alice 的第一个密码和签名，Alice 如果想造假抛出之前的交易，Bob 都可以一次取走通道里的所有比特币。

这样，闪电网络通过惩罚不诚实的企图造假方来保证大家彼此不会偷奸耍滑。所有的人都会在最新的当前交易状态合法签名并合法流转广播。

步骤 2：建立微支付交易通道（网络）

我们前面讲解了如何建立微支付的双向交易通道实现两个人之间的支付，现在如果 Alice 想要向第三个人 Carol 支付比特币该怎么办呢？

1）Alice 可以跟 Bob 一样建立与 Carol 之间的双向交易通道向 Coral 支付（当然建立通道需要成本）；

2）如果 Bob 刚好已经跟 Carol 建立了双向交易支付通道，则 Alice 可以通过已经建立的自己跟 Bob 的交易通道给 Carol 支付，走 Alice → Bob → Carol 通道。

对 Alice 而言，她的疑虑是怕 Bob 没把钱给 Carol，同时也怕 Carol 否认她收到 Bob 给的钱。

消除 Alice 顾虑的办法是：

1）确认 Bob 将钱给 Carol 后，才将钱给 Bob，然后知会 Carol，Bob 会转交钱给她。

2）Alice 要求 Carol 随机生成一个密码，将密码的哈希函数结果交给 Alice，并告知 Carol，只有 Bob 将钱给她后，才能将这个密码给 Bob。同时，Alice 告诉 Bob，只有 Bob 钱给到 Carol 之后，才能拿到 Carol 才知道的密码，之后交给 Alice 确认后，Alice 才会给 Bob 钱。因为 Bob 用比特币换到了只有 Carol 才知道的密码。

而对于 Bob 而言，他的担忧是：

1）他需要相信他把钱给 Carol 之后能拿到密码；

2）他还需要相信一旦他拿到密码 Alice 真会给他钱。

哈希时间锁合约（Hashed Time-Locked Contract，HTLC）可以解除 Bob 的担忧：Alice 建立一个 1 BTC 的多重签名合约。Case 1 对于 Bob，若合约中有他的签名以及正确的从 Carol 处得到的密码，即可解锁。Case 2 对于 Alice，使用 CLTV-Timelock 时间锁，确保自己的签名在一个约定的合同期有效，当 Bob 拿到密码就履行合约将钱给 Bob，同时广播让公众都知道，如果合同逾期，Bob 拿不到密码或提供不了自己的签名，Alice 用自己签名即可解锁 CLTV-Timelock 拿回自己的钱 1 BTC。

让我们想象一个这样的网络，不单单是 Alice 跟 Bob 之间建立这样的哈希时间锁定合约，Bob-Carol 之间也可以建立这样的 HTLC 合约，这样我们可以建立一个：Alice → Bob → Carol →……无数这样的节点构成了闪电网络[⊖]。

⊖ http://lightning.network.

步骤 3：完成微支付交易并关闭支付通道

至此，我们看到了闪电网络的巨大的能力，那就是前面我们所分析与描述的发生在闪电网络的交易，都不是必须要一笔对一笔地写到比特币区块链主链上，从而为比特币网络节省了很多消耗。

如果这个时候 Alice 与 Bob 想静悄悄地关闭彼此建立起来的支付通道，他们只要在主链上产生一笔交易，将交易通道开通直到交易结束，每个参与方拿到自己最后交易状态的份额，最后又回到主链上来。发生在通道里的所有交易隐私性就可以保护起来了。

我们熟悉的 Hyperledger Fabric 的通道（Channel）就是借鉴闪电网络的原理，并以此来作为保护交易对手的信息隐私。

其实当交易对手上方决定关闭微支付通道的时候回到比特币主链上，其实只需要告诉主链一个开通微支付通道合约交易和一个关闭微支付通道合约交易。期间交易对手们无论交易过多少次，对主链来说都无关紧要。这样，为比特币提供了一种不在主链上做交易的机制和解决方案。可以大大减轻比特币主链的性能瓶颈。

5.4　多链：区块链应用的扩展交互

现在我们看到的很多区块链基础技术构架都是单链形态。但在现实社会各个产业价值网络中，多链结构的技术更符合复杂价值逻辑的实际应用，各行各业或者说各个领域都有可能针对不同的业务场景去构造一条链，这些平行的链之间会存在数据交互的需求，即便是在同一个业务场景下，也有可能构建一组共同配合工作的链来完成复杂的业务逻辑，这个时候各个链之间的交互能力就会变得重要起来。我们在此提出了用跨链连接器连接多个可根据商业应用场景分别构建起来的价值链的多链架构理念。

如图所示，不同的链之间可以通过一个专门设计的跨链连接器进行互连，跨链连接器就类似于机械部件中的连接件，在软件领域中有个专门的术语叫“中间件”，在这样的一个中间件中可以定义大家共同遵循的数据接口规范，各种不同的链只要提供针对接口规范的接口实现，就可以进行互联，既实现了标准化，也确保了链本身设计的灵活性。它的工作方式如下：

1）主链作为总账本，分别在多个价值共识节点之间进行同步，注意，这里所说的主链是一个相对的概念；

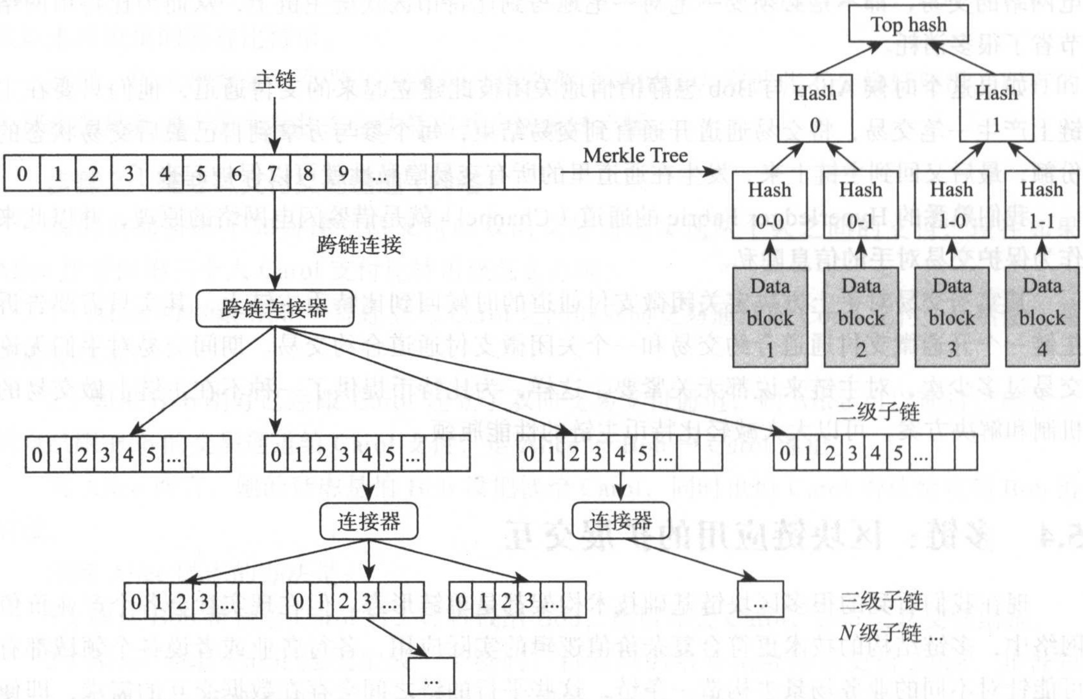
2）多链结构中，每一条链都是自成一个体系的，无论是主链还是子链，都在自己的节点之间进行数据的复制和传递，独立记账；

3）主链和子链间通过跨链连接器保证主链对子链价值的一致性、合法性、完整性，做跨链校验以及纠正和确认，保证链间价值合法合理合规流转；

4）主链和子链按照功能和价值网络搭建业务结构，并编写跨链逻辑的具体实现；

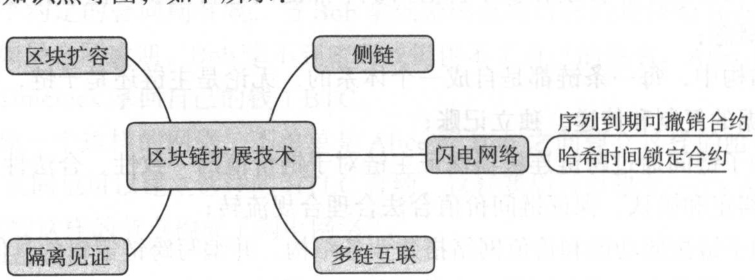
5）单一功能的账本使用和记录，在单独的功能子链上进行，在校验没有发生跟主链或

非本链数据变动的情况下，只在单链自行进行查询、校验、记账等单链需要行使的区块链价值网络体系功能，极大提高价值网络并行计算能力，维系良好的可扩展性和可利用性。



5.5 知识点导图

本章主要是对区块链系统目前出现的各种扩展技术进行了一个基本的介绍，主要为侧链、闪电网络以及多链互联。区块链系统由于其特有的分布式结构设计以及去中心化运行维护的特点，在软件功能升级、版本变更等事项的处理上也就有了特有的问题，各类扩展技术的出现，无疑是对于这些难题解决的一个破冰方案。当然这些技术方案本身也还有待时间去考量验证，无论如何，它们都是组成多彩缤纷的区块链技术体系的重要部分。我们来看下本章的知识点导图，如下所示：



第 6 章　Chapter 6

区块链开发平台：以太坊

6.1　项目介绍

6.1.1　项目背景

区块链技术是建立信任机制的技术，常常被认为是自互联网诞生以来最具颠覆性的技术。然而自从比特币诞生后，一直以来都没有很好的开发平台，想要借助于区块链技术开发更多的应用还是具有相当难度的，直接使用比特币的架构来开发则很复杂繁琐。事实上，比特币仅仅被设计为一个加密数字货币系统，只能算是区块链技术的一个应用，虽然也具备一些指令程序解析能力，但只是非常基础的堆栈指令，无法用来实现更广阔的业务需求。以太坊是目前使用最广泛的支持完备应用开发的公有区块链系统，本章我们就来介绍一下在该系统中应用的开发与部署方式。

与比特币相比，以太坊属于区块链 2.0 的范畴，是为了解决比特币网络的一些问题而重新设计的一个区块链系统。人们发现比特币的设计只适合加密数字货币场景，不具备图灵完备性，也缺乏保存实时状态的账户概念，以及存在 PoW 机制带来的效率和资源浪费的问题。最关键的问题是，在商业环境下，需要有高效的共识机制、具有图灵完备性、支持智能合约等多应用场景，以太坊在这种情况下应运而生。那么，以太坊被设计为一个什么样的系统呢？首先它是一个通用的全球性区块链，也就是说它属于公有链，这一点与比特币是一样的，并且可以用来管理金融和非金融类型的应用，同时以太坊也是一个平台和编程语言，包括数字货币以太币（Ether）以及用来构建和发布分布式应用的以太脚本，也就是智能合约编程语言。

比特币	以太坊
简单堆栈指令	合约编程语言

如图所示，这就是以太坊与比特币最大的一个区别，也因为提供了一个功能更强大的合约编程环境，使得用户可以在以太坊上编写智能合约应用程序，直接将区块链技术的发展带入到2.0时代。通过智能合约的设计开发，可以实现各种商业与非商业环境下的复杂逻辑，如众筹系统、数字货币、融资租赁资产管理、多重签名的安全账户、供应链的追踪监控等。通过智能合约的应用，可以将传统的软件系统链化，发挥出更强大的管理能力。理论上，我们可以在以太坊上实现一个比特币系统，而且实现过程相当简单，只需要编写一个符合比特币逻辑的智能合约就可以了。在这方面，以太坊平台相当于隐藏了底层技术的复杂性而让应用开发者更多地专注在应用逻辑及商业逻辑上。

以太坊的发展历史并不长，2013年年末，Vitalik Buterin（社区一般尊称他为V神），一位俄罗斯90后发布了以太坊的初版白皮书，项目就此启动了。之后的项目开展进度非常快，仅仅半年多时间就发布了5个版本的概念验证，充分体现了极客技术团队的效率和实力。大概是为了致敬比特币，团队开发所需费用直接接受的是比特币投资。值得一提的是，在开发过程中，以太坊设计了一个特有的叔区块的概念。我们知道在比特币中，一旦某个矿工挖矿成功，那么系统奖励的比特币就都是那个矿工的，其他矿工一无所获，而以太坊中将没有挖矿成功的矿工产生的废区块也纳入了奖励范畴，根据一定规则发放奖励。直至2015年7月，官方团队发布了正式的以太坊网络，一片新的天地就此打开。

以太坊在国内社区的发展也是如火如荼，为了方便国内用户更加方便快捷地同步以太坊区块数据，EthFans（国内最大的以太坊中文技术社区，网址为 http://ethfans.org）发起了星火节点计划。类似于比特币的种子节点，星火节点的信息会被打包到节点文件中，让社区成员自由下载，通过使用节点文件，本地运行的以太坊客户端可以连接到更多超级节点，大大加快了区块同步速度。我们看一下星火节点的浏览页面（页面的网址是 https://stats.ethfans.org）：

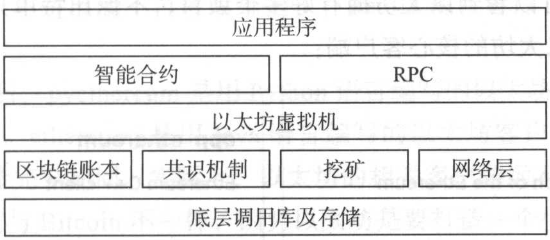


页面上列出了目前的星火节点名称，同时也显示了最新的区块高度、平均网络哈希速率等以太坊网络指标信息。

6.1.2 以太坊组成

以太坊的模块结构与比特币其实并没有本质的差别，还是那些物件，如区块链账本、共识机制、核心节点、P2P网络、可编程逻辑等，虽然很多细节（如区块的结构、数据编码方式、交易事务结构等）都有差别，但本质的特点是智能合约的全面实现，支持了全新的合约编程语言，以及为了运行合约增加了一个以太坊虚拟机。因此我们在理解以太坊的时候，基本上可以参照比特币的结构思路。如果说比特币是利用区块链技术开发的专用计算器，那么以太坊就是利用区块链技术开发的通用计算机，简单地说，以太坊 = 区块链 + 智能合约，开发者在以太坊上可以开发任意的应用，实现任意的智能合约。从平台的角度来讲，以太坊类似于苹果的应用商店；从技术角度来讲，以太坊类似于一个区块链操作系统。

我们来看一下以太坊的组成结构：



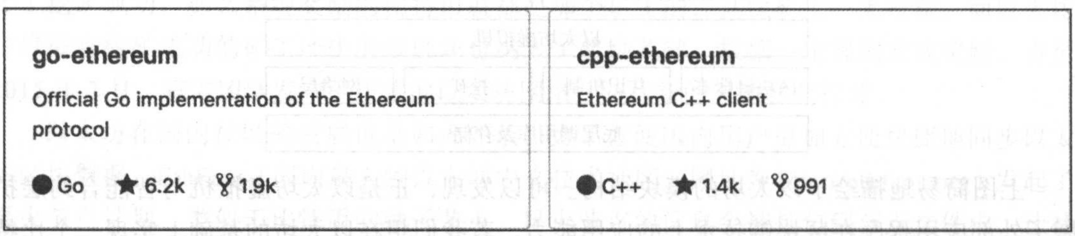
上图简易地描绘了以太坊的模块结构。可以发现，正是以太坊虚拟机与智能合约层扩展了外部应用程序在区块链技术上的应用能力。若我们想在以太坊的基础上实现一个比特币系统，只要在智能合约层开发一个与比特币逻辑一致的合约程序就可以了。当然只要你愿意，可以根据爱好或者需求去实现任何数字货币系统，它们都能通过以太坊网络良好地运行。值得注意的是，以太坊中的智能合约是运行在虚拟机上的，也就是通常说的EVM（Ethereum Virtual Machine，以太坊虚拟机）。这是一个智能合约的沙盒，合约存储在以太坊的区块链上，并被编译为以太坊虚拟机字节码，通过虚拟机来运行智能合约。由于这个中间层的存在，以太坊也实现了多种语言的合约代码编译，网络中的每个以太坊节点运行EVM 实现并执行相同的指令。

可能有些读者在这个环节一时不太理解，虽然看结构图是很简单，原理也是一目了然，可是细细一想，总觉得不够通透。如果说以太坊靠实现一个智能合约就能实现比特币，那岂不是说比特币就是一份合约？让我们来理一下这里的思路。首先比特币系统肯定不只是一份合约程序，只能说比特币的交易事务就是一份合约，比特币系统拥有自己的区块链账本、共识机制、挖矿系统等，这些基础结构都为一件事服务，就是运行比特币的智能合约：比特币交易事务。我们知道比特币之所以被称为可编程加密数字货币，就是因为其交易事务的结构中拥有锁定脚本和解锁脚本两段指令程序。从技术上来讲，比特币系统就是通过

执行交易事务中的锁定脚本和解锁脚本完成了比特币的发行和转账交易，也就是说比特币中的一切机制都是为了这一固定功能的合约而运行存在的。那么现在以太坊来了，大家觉得偌大一个系统，就只能运行一种智能合约，实在是太约束了。如果把锁定脚本和解锁脚本的编程能力加强，把交易事务的结构再扩展一下，使智能合约的能力不只是实现一个数字货币的转账交易，那就打开了另一片天地。不管是什么功能的合约，站在技术角度来讲，无非就是通过执行一组程序改变了一些值。我们不但可以实现数字货币，还可以实现众筹合约、担保合约、融资租赁合约、期货合约以及各种其他金融与非金融的订单合约，所有这些合约的执行都会被以太坊打包进区块，这样就实现了基于区块链的全功能智能合约。如果说比特币是二维世界的话，那么以太坊就是三维世界，可以实现无数个不同的二维世界。

现在让我们来更加具体地了解下以太坊，毕竟再怎么神奇强大，总归也就是一套软件，我们就来认识下以太坊具体的软件组件。

以太坊的源码是维护在 GitHub 上的，通过链接 https://github.com/ethereum 可以查看，在这个源码官网我们可以看到以太坊拥有好多个项目，不像比特币只有一个 Bitcoin，一目了然。我们先看一下以太坊的核心客户端：



可以看到，以太坊有两种语言版本的核心客户端：一个是 Go 语言版本，这也是官方首推的版本；另外一个是 C++ 语言的版本。两种版本的功能和使用是一样的，只不过用不同的语言实现，对于想要深入了解源码的读者，可以根据自己的语言偏好去下载对应的源码。除了核心客户端外，以太坊还提供了一系列其他独立使用的工具，比如新的实验性的合约编程语言 Viper、Solidity，以太坊的 JavaScript 调用库 Web3.js，以太坊官方钱包等。截至 2017 年 7 月，GitHub 官网上已经放了 100 多个各类功能的工具项目，我们整理一些常用的进行说明：

1）go-ethereum。官方的 Go 语言客户端，客户端文件是 geth。这是使用最广泛的客户端，类似于比特币的中本聪核心客户端，可用于挖矿、组建私有链、管理账号、部署智能合约等。但是注意不能编译智能合约（1.6 之前的版本还是内置编译模块的，1.6 之后就独立出去了）。该客户端可以作为一个独立程序运行，也可以作为一个库文件嵌入其他的 Go、Android 和 iOS 项目中，它没有界面，是一个命令行程序。

2）cpp-ethereum。与第一个一样，只不过是用 C++ 实现的。

3）EIP。EIP 描述以太坊平台标准，包含核心协议说明、客户端 API 以及合约标准等。

4）Mist 客户端。Mist 目前主要是钱包客户端，未来定义为一个 DAPP 市场交易客户端，类似于苹果市场。实际上 Ethereum Wallet 可以看作配置在 MistBrowser 上的一个应用，因此通常也叫 Mist/Ethereum Wallet。Mist 一般是配合 go-ethereum 或者 cpp-ethereum 运行的，如果在 Mist 启动的时候没有运行一个命令行的 ethereum 客户端，则 Mist 将启动区块链数据同步（使用绑定的客户端，通常默认是 geth，因此注意了，Mist 是会携带核心客户端的）。如果想要 Mist 运行在一个私有网络，只要在 Mist 启动前先启动节点（也就是 geth）即可，Mist 可以通过 IPC 连接到私有链。

5）Solidity 项目。Solidity 使用 C++ 开发，客户端文件为 solc，跨平台，使用命令行界面。solc 实际上是一个基本的编译平台，Solidity 是以太坊智能合约的编程语言。

6）browse-solidity 项目。browse-solidity 是智能合约浏览器版本的开发环境，可以直接在浏览器中进行开发、调试、编译。

7）Remix。Remix 是智能合约（以太坊称为 DAPP）的开发 IDE，采用图形化界面，可以支持智能合约（DAPP）的编写、调试、部署，是目前最主流的以太坊智能合约开发平台。之前还有个 Mix 项目，不过已经不再继续维护了，Remix 现在可以与 browser solidity 集成在一起使用了。

8）pyethereum 项目。pyethereum 是用 Python 语言编写的以太坊客户端。

9）ethereumj 项目。ethereumj 是用 Java 语言编写的以太坊客户端，与前面 Go 语言编写的客户端 geth 的功能完全相同。实际上，以太坊的相关客户端远不止这些，在 GitHub 站点上也能看到很多，这与 Bitcoin 不一样，因为以太坊是要打造一个生态。

6.1.3 关键概念

以太坊在开发时着重设计了虚拟机和智能合约相关的规范，这是以太坊的主要特点，然而作为一个开辟了区块链 2.0 智能合约时代的新平台，其特点以及改善之处远不止这些，在本节中，我们对以太坊中的一些关键概念做一些阐述。

1. 状态

状态的概念是在以太坊白皮书中提出的，我们先来截取以太坊白皮书中提及状态的几段文字描述：

以太坊的目标就是提供一个带有内置的成熟的图灵完备语言的区块链，用这种语言可以创建合约来编码任意状态转换功能。

从技术角度讲，比特币账本可以被认为是一个状态转换系统，该系统包括所有现存的比特币所有权状态和“状态转换函数”。状态转换函数以当前状态和交易为输入，输出新的状态。

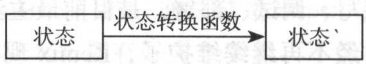
在标准的银行系统中，状态就是一个资产负债表，一个从 A 账户向 B 账户转账 X 美元的请求是一笔交易，状态转换函数将从 A 账户中减去 X 美元，向 B 账户增加 X 美元。如果

A 账户的余额小于 X 美元，状态转换函数就会返回错误提示。

比特币系统的“状态”是所有已经被挖出的、没有花费的比特币（技术上称为“未花费的交易输出”，unspent transaction outputs 或 UTXO）的集合。

一笔交易包括一个或多个输入和一个或多个输出。每个输入包含一个对现有 UTXO 的引用和由与所有者地址相对应的私钥创建的密码学签名，每个输出包含一个新的加入到状态中的 UTXO。

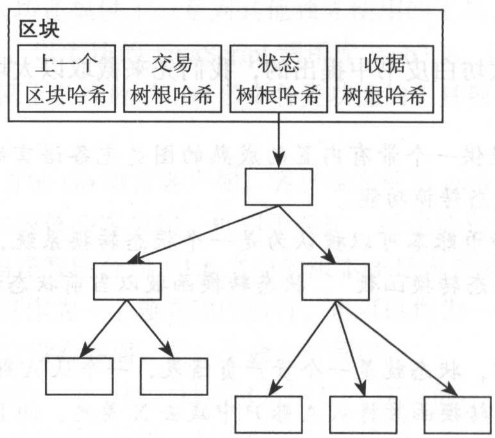
看到这里，不知道大家对状态的概念是否有一些感觉了。实际上以太坊是站在一个更高的维度来看待区块链账本中的数据变化。如果不发生任何交易事务，那相当于账本就是静态的，就好像是一个化学容器，里面有各种原材料，一旦发生了化学反应，不管是什么样的反应过程，反应结束后，容器中的状态肯定不一样。对于区块链账本，这里的变化可以是指一笔转账，也可以是合约的某个规则被激活等，总之就是数据动了，以太坊中将变化的过程称为状态转换函数，如下图所示：



通过这样一种更高层面的抽象，就使以太坊的设计具备了实现任意智能合约的基础，这里的状态数据可以是任何形式的（包括比特币那种 UTXO 的机制），状态函数也可以是任何过程的，只要符合业务需求即可，没有任何限制。

在以太坊的每一个区块头，都包含了指向三棵树的指针，分别是：状态树、交易树、收据树。交易树指针就类似于比特币区块头中的梅克尔树根，交易树是用来代表区块中发生的所有交易历史的；状态树代表访问区块后的整个状态；收据树代表每笔交易对应的收据，所谓的收据是指每一笔交易影响的数据条，或者说是每一笔交易影响的结果。这些都是针对比特币中单一的梅克尔交易树的增强，通过状态树可以很方便地获得类似账户存在与否、账户余额、订单状态这样的结果，而不用只依靠交易事务去追溯。

我们看一下以太坊中状态树的示意图：



如图所示，在状态树中存储了整个系统的状态数据，如账户余额、合约存储、合约代码以及账户随机数等数据。我们在玩游戏的时候，有个功能叫存档，可以把当时的各项游戏数据都记录下来，状态树在功能效果上与此类似。

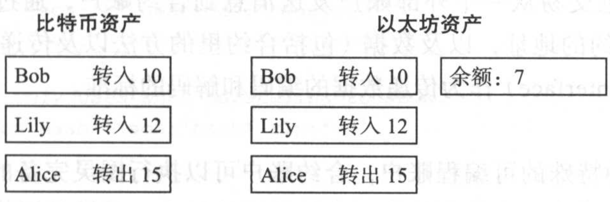
再来看一下以太坊源码中，是如何在区块头中定义这个状态树根哈希的：

```
// 定义以太坊区块链中的区域头结构
type Header struct {
	ParentHash  common.Hash    `json:"parentHash"       gencodec:"required"`
	UncleHash   common.Hash    `json:"sha3Uncles"       gencodec:"required"`
	Coinbase    common.Address `json:"miner"            gencodec:"required"`
	Root        common.Hash    `json:"stateRoot"        gencodec:"required"`
	TxHash      common.Hash    `json:"transactionsRoot" gencodec:"required"`
	ReceiptHash common.Hash    `json:"receiptsRoot"     gencodec:"required"`
	Bloom       Bloom          `json:"logsBloom"        gencodec:"required"`
	Difficulty  *big.Int       `json:"difficulty"       gencodec:"required"`
	Number      *big.Int       `json:"number"           gencodec:"required"`
	GasLimit    *big.Int       `json:"gasLimit"         gencodec:"required"`
	GasUsed     *big.Int       `json:"gasUsed"          gencodec:"required"`
	Time        *big.Int       `json:"timestamp"        gencodec:"required"`
	Extra       []byte         `json:"extraData"        gencodec:"required"`
	MixDigest   common.Hash    `json:"mixHash"          gencodec:"required"`
	Nonce       BlockNonce     `json:"nonce"            gencodec:"required"`
}
```

这段代码可以在以太坊源码中的 go-ethereum/core/types/block.go 文件中找到，这是一个自定义结构类型，定义了以太坊中的区块头结构，可以看到：其中有个属性 Root，是 common.Hash 类型，说明这是一个哈希值，并且是 stateRoot（状态树根哈希）。除了这些，我们同样能看到有 TxHash 和 ReceiptHash，分别对应了交易树根哈希和收据树根哈希。

2. 账户

在以太坊系统中，状态是由被称为"账户"的对象和在两个账户之间转移价值和信息的状态转换构成的，每个账户有一个 20 字节的地址，这个其实就跟银行账户差不多意思，在比特币中是没有账户这个概念的，或者说比特币中只有状态转换的过程历史。这里我们再来对比一下比特币，假设 Alice 既使用比特币也使用以太坊，并且初次使用，之前没有余额，那么 Alice 在两者中的账本信息大概是这样的：



可以看到，以太坊中由于具备账户的概念，可以直接获得当前的余额，这个余额相当于 Alice 资产当前的状态，而比特币中只有流水账，要获得当前余额，只能通过计算获得。

我们来具体了解一下以太坊中的账户，既然是账户就应有账户结构，通常包含下面 4 个部分。

1）随机数，用于确定每笔交易只能被处理一次的计数器，实际上就是每个账户的交易计数，用以防止重放攻击，当一个账户发送一笔交易时，根据已经发送的交易数来累加这个数字，比如账户发送了 5 个交易，则账户随机数是 5。

2）账户目前的以太币余额。

3）账户的存储（默认为空）。

4）账户的合约代码（只有合约账户才有，否则为空）。

这些其实就是以太坊源码中的定义，我们常常说要按图索骥，寻踪觅迹，任何一个定义，一个逻辑，我们都要看看它的来源到底是什么样的，我们来看看账户在以太坊源码中的定义描述：

```
// 以太坊中的账户对象结构定义
// 这些数据对象会存储在以太坊中的梅克尔树中
type Account struct {
    Nonce     uint64
    Balance   *big.Int
    Root      common.Hash // merkle root of the storage trie
    CodeHash  []byte
}
```

可以看到，账户结构的定义中与上述的四项属性一一对应，源码中就是这么定义的，其中的 Root 也就是所谓的账户存储空间，是一个根哈希值，指向的是一棵 patricia trie（帕夏尔前缀树），关于 patricia trie 的概念在下面会有介绍，总之就是一种存储结构，类似梅克尔树，但更复杂一些。

以太坊中的账户是区分类型的。

（1）外部账户

外部所有账户，术语叫 EOA，全称是 Externally Owned Account，这个就是一般账户的概念。外部所有账户是由一对密钥定义的，一个私钥一个公钥，公钥的后 20 位作为地址，这个跟比特币中的公私钥以及钱包地址类似。外部所有账户是没有代码的，但是可以通过创建和签名一笔交易从一个外部账户发送消息到合约账户，通过传递一些参数，比如 EOA 的地址、合约的地址，以及数据（包括合约里的方法以及传递的参数），使用 ABI（Application Binary Interface）作为传递数据的编码和解码的标准。

（2）合约账户

合约账户是一种特殊的可编程账户，合约账户可以执行图灵完备的计算任务，也可以在合约账户之间传递消息，合约存储在以太坊的区块链上，并被编译为以太坊虚拟机字节

码，合约账户也是有地址的，不过与外部所有账户不同，不是根据公钥来获得的，而是通过合约创建者的地址和该地址发出过的交易数量计算得到。

我们可以看到，外部所有账户在以太坊中就相当于一把钥匙，合约账户则相当于一个机关，一旦被外部所有账户确认激活，机关就启动了。

3. 交易

以太坊中的交易，也就是状态一节中所说的转换过程。通常提到交易，大家都会习惯性地认为是转账交易这种意思，在以太坊中交易的概念是比较广义的，因为以太坊并不仅仅支持转账交易这样的合约功能，它的定义如下：在以太坊中是指签名的数据包，这个数据包中存储了从外部账户发送的消息。所谓的交易就是一个消息，这个消息被发送者签名了，如果类比一下比特币的话，可以发现比特币中的交易也在这个范畴内，在比特币中也是通过转账发起者签名了一个 UTXO 数据然后发送出去，只不过比特币中只能发送这种固定格式的消息，源代码中写死了。

我们来看一下以太坊中的交易格式是什么，先来看下源码中的定义：

```
type Transaction struct {
    data txdata
    // caches
    hash atomic.Value
    size atomic.Value
    from atomic.Value
}
```

在这个定义中，最主要的就是 data 字段，这是一个命名为 txdata 的结构类型字段，代表了真正的交易数据结构，其余三个都是缓冲字段，我们来看一下 txdata 的定义：

```
type txdata struct {
    AccountNonce uint64          `json:"nonce"    gencodec:"required"`
    Price        *big.Int        `json:"gasPrice" gencodec:"required"`
    GasLimit     *big.Int        `json:"gas"      gencodec:"required"`
    Recipient    *common.Address `json:"to"       rlp:"nil"` // nil means
contract creation
    Amount       *big.Int        `json:"value"    gencodec:"required"`
    Payload      []byte          `json:"input"    gencodec:"required"`

    // Signature values
    V *big.Int `json:"v" gencodec:"required"`
    R *big.Int `json:"r" gencodec:"required"`
    S *big.Int `json:"s" gencodec:"required"`

    // This is only used when marshaling to JSON.
    Hash *common.Hash `json:"hash" rlp:"-"`
}
```

以下是一些说明。

1）AccountNonce：表明交易的发送者已发送过的交易数，与账户结构中定义的随机数对应。

2）Price 与 GasLimit：这是以太坊中特有的概念，用来抵抗拒绝服务攻击。为了防止在代码中出现意外或有意无限循环或其他计算浪费，每个交易都需要设置一个限制，以限制它的计算总步骤，说白了就是让交易的执行带上成本，每进行一次交易都要支付一定的手续费，GasLimit 是交易执行所需的计算量，Price 是单价，两者的乘积就是所需的手续费，交易在执行过程中如果实际所需的消耗超出了设置的 Gas 限制就会出错回滚，如果在范围内则执行完毕后退还多余的部分。

3）Recipient：接收方的地址。

4）Amount：发送的以太币金额，单位是 wei。

5）Payload：交易携带的数据，根据不同的交易类型有不同的用法。

6）V、R、S：交易的签名数据。

可能有些读者会有疑问，通过这个交易结构，怎么看出是谁发出的呢，为什么只有接收方的地址却没有发出方的地址呢？那是因为发送者的地址可以通过签名获得。

我们提到了不同的交易类型，那么在以太坊中都有哪些不同的交易类型呢？接下来我们就一一说明一下，为了让差别一目了然，我们通过 Web3.js 的调用格式来说明。Web3.js 是一个 JavaScript 库，可以通过 RPC 调用与本地节点通信，实际上就是一个外部应用程序用来调用以太坊核心节点功能的一个调用库。

（1）转账交易

以太坊本身内置支持了以太币，因此这里说的转账就是指从一个账户往另一个账户转账发送以太币，我们知道要转账，一般来说得要有发送方、接收方、转账金额。指令格式如下：

```
web3.eth.sendTransaction({from:"",to:"",value:});
```

from 后面是发送方的账户地址，to 后面是接收方的地址，value 后面是转账金额。

（2）合约创建交易

有读者可能会感到奇怪，创建一份合约怎么也是交易，又没有向谁转账，我们再次重申一下以太坊中交易的定义：在以太坊中，交易是指签名的数据包。不过，合约在创建的时候是需要消耗以太坊的，从这个层面来看，也算是一种传统的交易吧。我们来看一下指令格式：

```
web3.eth.sendTransaction({from: "",data: ""});
```

from 后面是合约创建者的地址，data 后面是合约程序的二进制编码，这个还是容易理解的。

（3）合约执行交易

合约一旦部署完成后，就可以调用合约中的方法，也就是执行合约，在以太坊中执行

合约也属于一种交易，我们来看一下合约执行交易的指令格式：

```
web3.eth.sendTransaction({from: "",to: "",data: ""});
```

from 后面是合约调用者的地址，to 后面是合约的地址，data 后面是合约中具体的调用方法以及传入的参数。实际上，转账交易也属于一种合约执行交易，只不过以太币是以太坊内置的数字货币，对于以太币的合约处理是系统直接自动完成的，不再需要指定一个合约。

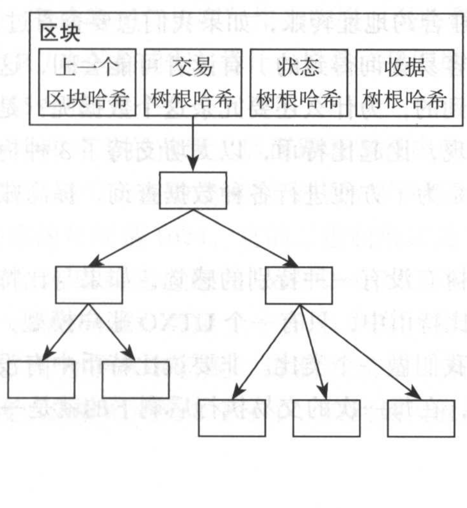
以上就是以太坊中的 3 种交易类型，在后续的章节中会有具体的操作示例，现在我们只要有个基本了解就行了。使用过比特币的朋友都知道，比特币是有很多计量单位的，从最小的“聪”到最大的“BTC”，那么以太坊中涉及以太币的交易计量单位有哪些呢，我们来说明一下。

以太币（Ether 币）的最小单位是 wei，类似于比特币中的最小单位是聪，然后每 1000 个递进一个单位，如下所示：

- kwei=1000 wei
- mwei=1000 kwei
- gwei=1000 mwei
- szabo=1000 gwei
- finney=1000 szabo
- ether=1000 finney

通过以上的换算关系，我们可以发现，1ether=1000 000 000 000 000 000wei，足有 18 个 0，可别看眼花了，我们使用命令行工具访问以太坊节点时，默认的以太币计量单位是 wei，如果是图形界面的钱包客户端，则一般是 ether，大家在使用时，一定要看清楚计量单位。

交易数据在以太坊区块中也是有棵树的，在上述介绍状态时，我们看过区块头的数据结构，其中就有一个交易树根哈希，交易树的概念与比特币中的梅克尔树是一个意思，只不过存储的结构与编码方式有些差别。



4. 收据

收据这个概念也是以太坊中特有的，字面的意思是指每条交易执行所影响的数据条，在以太坊的区块头中存储有收据树的根哈希值，也就是说在每个区块中，收据和交易以及状态一样，都是一棵树，那么收据中到底是些什么呢？我们还是看一下源码中怎么定义的：

```
// 收据对象描述的是交易事务产生的结果
type Receipt struct {
    // Consensus fields
    PostState         []byte      `json:"root"`
    CumulativeGasUsed *big.Int   `json:"cumulativeGasUsed" gencodec:"required"`
    Bloom             Bloom       `json:"logsBloom"         gencodec:"required"`
    Logs              []*Log      `json:"logs"              gencodec:"required"`

    // Implementation fields (don’t reorder!)
    TxHash          common.Hash    `json:"transactionHash" gencodec:"required"`
    ContractAddress common.Address `json:"contractAddress"`
    GasUsed         *big.Int       `json:"gasUsed" gencodec:"required"`
}
```

乍看之下有点不明所以，不过既然是指交易执行后的影响结果，那就跟交易有关，交易执行后会影响状态的变更，会消耗 Gas，我们来看一看结构定义中的主要属性。

1）PostState：这是状态树的根哈希，不过不是直接存储的哈希值，而是转换为字节码存储，通过这个字段使得通过收据可以直接访问到状态数据。

2）CumulativeGasUsed：累计的 Gas 消耗，包含关联的本条交易以及之前的交易所消耗的 Gas 之和，或者说是指所在区块的 Gas 消耗之和。

3）TxHash：交易事务的哈希值。

4）ContractAddress：合约地址，如果是普通的转账交易则为空。

5）GasUsed：本条交易消耗的 Gas。

我们可以看到，收据实际上是一个数据的统计记录，记录了交易执行后的特征数据，那么，这个数据保留下来有什么用呢？主要还是方便取得某些统计数据，比如我们创建了一个众筹合约，大家可以往合约地址转账，如果我们想要查看过去 20 天内这个合约地址的众筹情况，通过收据是很容易查询得到的。有读者可能会问，这样的查询就算没有收据这种数据的存在也是可以得到的，为什么还要冗余这个数据呢？是的，技术上来说，收据确实不是必需的，我们也发现，比起比特币，以太坊支持了 3 种梅克尔树：交易树、状态树和收据树。其目的无非就是为了方便进行各种数据查询，提高账本数据在各种需求之下的统计查询效率。

不知道大家对于收据树有没有一种特别的感觉，如果与比特币相比，我们发现，特别像比特币中的 UTXO，在比特币中，只有一个 UTXO 账户模型，当然严格来说比特币是没有账户的，只不过在这里我们做一个类比。非要说比特币中有没有账户的话，UTXO 数据就是比特币中的账户模型，在每一次的交易执行后剩下的就是一个 UTXO 的结果，以太坊

中的收据与这个很相像，它也是交易执行后的一个结果，而且收据与交易是关联对应的，这与 UTXO 的输出对应输入也是异曲同工的，就个人的技术倾向，实际上 UTXO 这种模型是非常可靠的，基本上不会发生数据不一致问题，读者可以反复体会一下。

5. RLP 编码

RLP（recursive length prefix），直译过来叫“递归长度前缀”，相当拗口的一个名词，相信第一次看见这个称呼的读者肯定很迷茫，总之这是一种数据编码方式。这种编码方式在以太坊中使用很普遍，是以太坊中对象序列化的主要方式，在区块、交易、账户状态等地方都有使用，比如交易数据从一个节点发送到另一个节点时，要被编译为一种特别的数据结构，这种结构称为 trie 树（也叫前缀树），然后根据这棵前缀树计算出一个根哈希（上述介绍的状态树、交易树、收据树都是这种方式），而这棵树中的每一个数据项都会使用 RLP 的方式编码。关于 trie 树的细节稍后再详谈，这里不再赘述。

既然是一种编码方式，那就好描述了，我们知道计算机中的数据在本质上都是二进制码，而编码方式就是一种约定的规则，将二进制数据通过某种格式要求进行组装，以便于数据传输的编码与解码，接下来我们就来了解下 RLP 的编码规则，看看它这个拗口的名字到底是怎么来的。

（1）单字节数据编码

对于单字节数据，如果表示的值的范围是 [0x00,0x7f]，则它的 RLP 编码就是本身，这个范围的数据其实就是 ASCII 编码，不过要注意的是，这里说的是单字节，如果一个数据的值虽然属于 ASCII 编码的值的范围，但却不是单字节的，那就不符合这个规则了，而要使用下面的规则。

（2）字符串长度是 0 ～ 55 字节

RLP 编码包含一个单字节的前缀，后面跟着字符串本身，这个前缀的值是 0x80 加上字符串的字节长度。由于被编码的字符串最大的字节长度是 55=0x37，因此单字节前缀的最大值是 0x80+0x37=0xb7，即编码的第一个字节的取值范围是 [0x80, 0xb7]。这个很好理解，就是第一个字节是“0x80+ 字符串字节长度”作为前缀，至此我们就理解了 RLP 中长度前缀的意思，至于 RLP 中的 R（也就是递归）是什么意思，我们接着往下看。

（3）字符串长度大于 55 字节

它的 RLP 编码包含一个单字节的前缀，后面跟着字符串的长度，再跟着字符串本身。这个前缀的值是 0xb7 加上字符串长度的二进制形式的字节长度，说得有点绕，举个例子就明白了，例如一个字符串的长度是 1024，它的二进制形式是 10 000 000 000，这个二进制形式的长度是 2 个字节，所以前缀应该是 0xb7+2=0xb9，字符串长度 1024=0x400，因此整个 RLP 编码应该是 \xb9\x04\x00 再跟上字符串本身。编码的第一个字节即前缀的取值范围是 [0xb8, 0xbf]，因为字符串长度二进制形式最少是 1 个字节，因此最小值是 0xb7+1=0xb8，字符串长度二进制最大是 8 个字节，因此最大值是 0xb7+8=0xbf。

注意在这种情况下，字符串前面是一个单字节的前缀以及字符串的长度，多了一个字符串长度也要跟着。单字节的前缀是指 0xb7 加上字符串长度的二进制形式的（如上述字符串长度是 1024 字节，则 1024 的二进制形式为 10 000 000 000，长度是 2 个字节，所以是 0xb7+2=0xb9，后面再跟上字符串的长度，1024 字节长度的 16 进制是 0x400）。

以上都是对于字符串的编码，接下来我们来看看列表的编码，难度稍微增加些，其实列表编码就是在上述的编码基础上进行的，列表中包含不止一个字符串，每个字符串的编码方式都是一样的，只不过在整体编码上有些差别。

（4）列表总长度为 0~55 字节

列表的总长度是指它包含的项的数量加上它包含的各项的长度之和，它的 RLP 编码包含一个单字节的前缀，后面跟着列表中各元素项的 RLP 编码，这个前缀的值是 0xc0 加上列表的总长度。编码的第一个字节的取值范围是 [0xc0, 0xf7]。

（5）列表总长度大于 55 字节

RLP 编码包含一个单字节的前缀，后面跟着列表的长度，再跟着列表中各元素项的 RLP 编码，这个前缀的值是 0xf7 加上列表总长度的二进制形式的字节长度。编码的第一个字节的取值范围是 [0xf8, 0xff]。

通过列表的编码规则，我们可以看到这里有递归的影子，除了前缀以外，其中的编码都是不断地重复单个字符串的编码方式对每一个列表项进行编码，这就是递归前缀编码的称呼来源。

（6）示例

- 字符串："dog" = [0x83, 'd', 'o', 'g']
- 列表：["cat", "dog"] = [0xc8, 0x83, 'c', 'a', 't', 0x83, 'd', 'o', 'g']
- 空字符串："" = [0x80]
- 空列表：= [0xc0]
- 整数：15 ('\x0f') = [0x0f]

读者可以根据对规则的理解，尝试编写一个 RLP 编码程序，体验一下这种编码的特点。

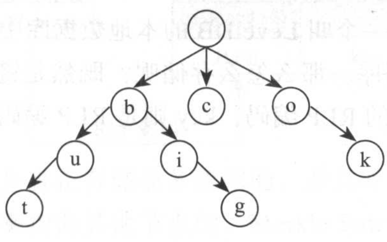
6. 梅克尔 – 帕特里夏树

我们知道，在比特币系统中有一个梅克尔树（Merkle Tree）的概念，在每一个区块头都有一个梅克尔根，实际上就是一个区块中交易哈希树的根哈希值，而以太坊中也有类似的结构，通过上述章节的学习，我们知道在以太坊的区块头中有 3 个根哈希，分别是状态树、交易树和收据树的根哈希，对应着各自的树结构，那么这些树结构与比特币中的梅克尔树有什么差别？严格来说，比特币中的梅克尔树叫二叉梅克尔树，以太坊中的则是梅克尔 – 帕特里夏树（有时也称为帕夏尔树），是一种更加复杂的结构，英文全称为 Merkle Patricia Tree，就是梅克尔树与帕特里夏树（以下以其英文名 Patricia Tree 称呼）的结合。

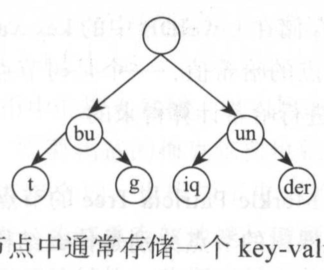
（1）Patricia Tree

我们来了解一下它的概念，以及在以太坊中到底如何应用。以交易数据为例，当交易数据从一个节点发送到另一个节点的时候，必须被编译为一个特别的数据结构，称为 trie（前缀树），然后计算生成一个根哈希。值得注意的是，这个 trie 中的每一个项都使用 RLP 编码（这就是 RLP 编码的一个应用场合了）。注意，在 P2P 网络上传输的交易是一个简单的列表，它们被组装成一个叫作 trie 树的特殊数据结构来计算根哈希，这意味着交易列表在本地以 trie 树的形式存储，发送给客户端的时候序列化成列表。实际上，在以太坊中，使用的是一种特殊的 trie 结构，也就是 Patricia Tree。这下我们明白了，比特币中是将交易数据组装成一棵二叉树然后计算根哈希，而以太坊中则是组装成一棵 Patricia Tree 然后计算根哈希。因此我们只要理解什么叫 Patricia Tree 就可以了，梅克尔哈希的计算没什么特别的。

大家在平时看一些资料的时候，看到以太坊关于 Patricia Tree 的介绍时，常常会看到 trie 这个名字，一会儿是 Patricia Tree，一会儿是 Patricia Trie 等，让人不明所以，我们先把这些名词称呼理一理。Patricia Tree 也称为 Patricia Trie、radix tree 或者 crit bit tree，是基于 trie tree 的一种结构，trie tree 是一种单词查找树结构，我们看下示例图：



trie 中每个节点存储单个字符，我们可以看到，but 与 big 两个单词共享了同一个前缀 b，通过这种方式可以节约存储空间，用通常的数组或者 key-value 键值对的方式都不能很好地节约存储空间，trie 的这种方式还为检索数据带来了便利，只要定位一个前缀，所有具有同一个前缀的数据都在一起了。然而，我们说 Patricia Tree 是基于 trie tree 的一种结构，但并不相同，在 trie tree 中通常每个节点只存储单个字符，而 Patricia Tree 的每个节点可以存储字符串或者说二进制串，这样就使得 Patricia Tree 可以存储更为一般化的数据，而不只是一个单词字符，如下图所示：



在这样的树结构中，每个节点中通常存储一个 key-value 键值对数据，key 用来保存索引，是用来搜索定位的，value 则是节点中具体的业务数据，key-value 是典型的字典数据结

构，因此这种结构也称为字典树，顾名思义，就是方便用来像查字典一样检索数据的结构。实际上我们日常生活中经常会用到这样的结构，抛开这些技术上的概念，比如我们在查新华字典的时候，通过拼音来查字，比如“海”字，我们会先翻到“h”开头的目录，可以发现有很多“h”开头的字，接着往下查“ha”开头的，还是有很多，最后查到“hai”，定位到“海”这个字了，这其实就是前缀索引树的应用，所以说很多看起来复杂的技术，在生活中其实都有在运用，不但艺术来源于生活，技术也是来源于生活的。

好了，到这里就可以结束了吗？答案是：没有！很不幸，以太坊中的树结构在这个基础上还要复杂不少，理解起来颇费周折。我们来看看以太坊中的 Merkle Patricia Tree 具体是哪种结构。

我们知道在一棵树结构中，无论树的结构有什么特别的，总归就是一个个的节点，事实上，以太坊中对节点还进行了不同类型的划分，分为空节点、叶子节点、扩展节点、分支节点，我们先不管这些节点分别有什么作用，只要知道节点中是 key-value 格式的数据存储格式就行了。

```
Node=(Key,Value)
```

这些节点数据会被存储在一个叫 LevelDB 的本地数据库中，LevelDB 是 Google 实现的一种非常高效的键值存储数据库。那么怎么存储呢？既然是键值存储，那就是有一个 key，有一个 value，value 就是节点的 RLP 编码，key 则是 RLP 编码的哈希值。

```
value=RLP(Node)
key=sha3(value)
```

如上所示，存储到 LevelDB 中的 value 是节点数据的 RLP 编码，而 key 则是这个 RLP 编码的哈希值，以太坊中使用了 SHA3 算法计算了哈希值，SHA3 是第三代 sha 哈希计算算法。以太坊网络中的核心客户端会不断地同步更新这个数据库以保持与网络中的其他客户端数据同步，我们看一下源码中的定义：

```
type SyncResult struct {
    Hash common.Hash
    Data []byte
}
```

SyncResult 是定义用来同步存储在 LevelDB 中的 key-value 数据的，显而易见，这里定义了两个字段类型：一个是树节点的哈希值，一个是树节点中包含的数据，而树节点的哈希值是通过对树节点的包含数据进行哈希计算得来的。

（2）节点类型

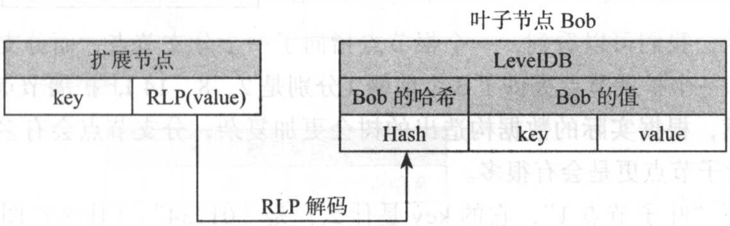
至止，我们知道了以太坊中 Merkle Patricia Tree 的节点是存储在本地的 LevelDB 数据库中的，接下来解析一下组成这棵树的节点分别是什么结构，刚才提到节点是有不同的类型的，那么分别有哪些类型？

1）空节点：表示空的意思，value 中是一个空串；

2）叶子节点：表示为 [key,value] 的一个键值对，其中 value 是数据项的 RLP 编码，key 是 key 数据的一种特殊的十六进制编码，叶子节点用来存储业务数据，叶子节点下面不再有子节点。

叶子节点	
key	RLP(value)

3）扩展节点：也是 [key,value] 的一个键值对，但是这里的 value 是指向其他节点的哈希值。什么哈希值呢？就是上面所说的存储在 LevelDB 中的节点哈希值，通过这个哈希值可以直接定位到某一个节点，也就是说扩展节点相当于一个指针节点。另外，扩展节点的 key 也被编码为一个特殊的十六进制编码。我们看以下示意图，图中的叶子节点 Bob 只是一个假设称呼，可以看到扩展节点的 value 部分实际上存储的是另外的节点的哈希值，通过这样的对应关系，可以使用扩展节点连接到另外一个节点。

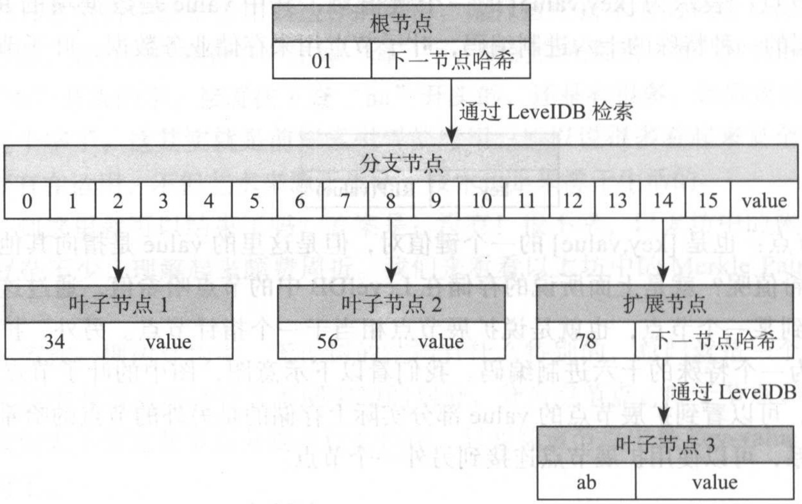


4）分支节点：叶子节点是真正存储业务数据的，并且叶子节点不再有子节点（要不怎么叫叶子呢），扩展节点是用来指向其他节点的。Merkle Patricia Tree 作为一种前缀树，主要特点是依靠共享的前缀来提高树结构的处理性能，那么这个前缀就很重要了，对于扩展节点和叶子节点来说，节点的 key 就是起到前缀的作用。通过上面的了解，我们知道叶子节点和扩展节点的 key 都会被编码为一种十六进制的格式，先不细究到底是什么样的格式，有一点我们知道，既然是十六进制的数据，那编码字符的范围就是 0 ～ F。如果需要一个节点的 key 能够包含所有这些字符的范围，则需要一个长度为 16 的列表，再加上一个 value，这样的节点类型称为分支节点，所以分支节点是一个长度为 17 的列表，我们看下分支节点的示例样式：

分支节点																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	value

5）树结构示例：比起比特币中的梅克尔树，以太坊中的设计复杂了很多。当然，由于比特币中仅支持转账交易合约，需要构造的梅克尔树也就是一棵二叉哈希树，自然是简单很多，以太坊中支持更广泛的智能合约，也增加了更多的概念，如上所述的账户、状态、收据等，数据种类复杂许多，为了能够更有效地进行增删改查操作，并且让树的结构更加平衡有效，因此设计出了许多有趣的结构，我们来看下这些节点类型组合起来会是怎样一

个效果：



如上图所示，我们可以看到，一个根节点指向了一个分支节点，而分支节点又为下面两个叶子节点和一个扩展节点提供了3个前缀（分别是2、8、14），扩展节点又指向了一个叶子节点。当然，根据实际的数据构造出的树会更加复杂，分支节点会有多个，扩展节点也会有多个，叶子节点更是会有很多。

我们看一下“叶子节点1”，它的key是什么？是“01234”。（什么？图中标记的不是34吗？）我们先来看下这个“01234”是怎么来的，首先从根节点的“01”开始，然后经过了分支节点的“2”，再到达自己的“34”，连起来就是“01234”，那么“叶子节点1”中的“34”是什么呢？这个其实是“叶子节点1”中key的尾缀部分，在帕特里夏树中就是依靠这样的前缀路径索引来定位到目标节点的。除了“叶子节点1”，其余的“叶子节点2”、“叶子节点3”以及“扩展节点”也是同样的索引逻辑。

大家观察这棵树的结构，可以发现这些节点类型的存在，就是要通过共享前缀的方式来充分提高存取效率，而且树的结构比较紧凑均衡，下图为各节点的key列表：

节点	key
根节点	01
叶子节点 1	01234
叶子节点 2	01856
扩展节点	011478
叶子节点 3	011478ab

（3）十六进制前缀

事情到这里似乎可以结束了，然而以太坊中的帕特里夏树还有一个特征，这个特征很有意思，如我们在前面看到的，分支节点的结构很特殊，是一个长度为17的列表，很容易

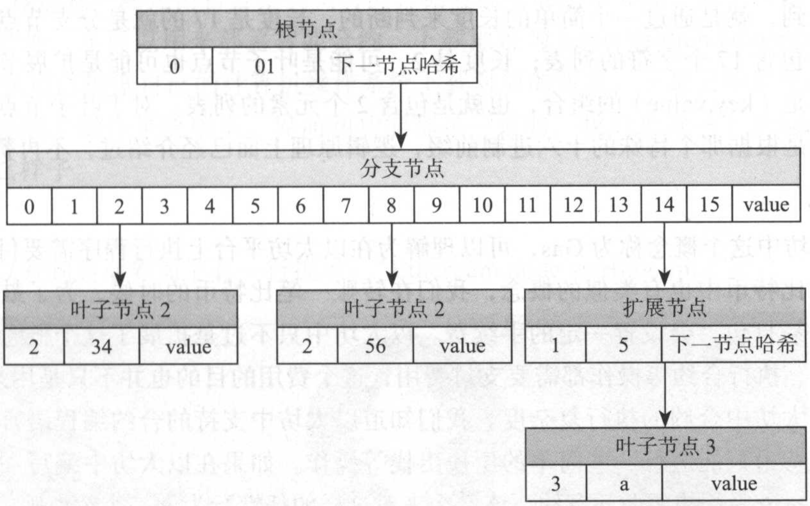
判断出来，但是扩展节点和叶子节点的长度都是2（节点的类型判断在下一节中有详细描述），那么对于都是具备（key,value）特征的叶子节点和扩展节点，怎么去区分呢？很简单，那就是在这两种节点的key部分增加一个前缀，一个十六进制字符的前缀，通过这个前缀字符用来表示节点是叶子还是扩展，除了用来判断类型外，还顺便用来编码表示key长度的奇偶性，具体如下：

1）十六进制长度的字符使用4位二进制表示，也就是半个字节，在这个4位二进制码中，最低位用来表示key长度的奇偶性，第二低位用来表示是否终止（1表示终止，也就是叶子节点，0表示扩展节点）。

2）这个半字节的字符由如下4种编码组成：

节点类型	key 长度奇偶性	前缀字符（十六进制）	前缀字符（二进制）
扩展节点	偶数	0	0000
扩展节点	奇数	1	0001
叶子节点	偶数	2	0010
叶子节点	奇数	3	0011

我们来看以下示例图：



注意 增加这个特殊的十六进制前缀并不是属于节点key的一部分，而仅仅是在构建树结构的时候附加上去的，我们知道整棵树表示的数据在网络中传递的时候就是一个列表数据，而树结构是以太坊客户端接收到数据后另行构造出来的。

（4）节点类型判断

补充一点，上述那些节点类型在以太坊中是怎么判断的呢？我们来看段ethereumjs中的源码片段（ethereumjs是以太坊的JavaScript模拟项目，其中实现的逻辑与以太坊是一致

的，方便测试使用)。

```
/*
 * Determines the node type
 * Returns the following
 * - leaf - if teh node is a leaf
 * - branch - if the node is a branch
 * - extention - if the node is an extention
 * - unknown - if somehting fucked up
 */
function getNodeType (node) {
  if (node.length === 17) {
    return 'branch'
  } else if (node.length === 2) {
    var key = stringToNibbles(node[0])
    if (isTerminator(key)) {
      return 'leaf'
    }

    return 'extention'
  }
}
```

可以看到，就是通过一个简单的长度来判断的，长度是17的就是分支节点，因为分支节点是一个包含17个字符的列表；长度是2，可能是叶子节点也可能是扩展节点，因为这两个节点都是（key,value）的组合，也就是包含2个元素的列表。对于叶子节点和扩展节点的区分，就是根据那个特殊的十六进制前缀，逻辑原理上面已经介绍过，不再赘述。

7. 燃料

在以太坊中这个概念称为Gas，可以理解为在以太坊平台上执行程序需要付出的成本或手续费。在比特币中也有类似的概念，我们在转账一笔比特币的时候，为了鼓励矿工尽快将我们的交易打包，会设置一定的手续费。以太坊中只不过是扩展了这个概念，在以太坊中创建合约、执行合约等操作都需要支付费用，这个费用的目的也并不只是用来激励矿工，还能约束以太坊中合约的执行复杂度。我们知道以太坊中支持的合约编程语言是图灵完备的，不像比特币只能进行一些简单的压栈出栈等操作。如果在以太坊中编写一个步骤很复杂，甚至是一个恶意的死循环合约，该怎么来对这样的任性行为做一个约束呢？那就是Gas的作用了，Gas是通过以太坊中合约的执行计算量来决定的，这个计算量可以简单地认为是算力资源的消耗，比如执行一次SHA3哈希计算会消耗20个Gas，执行一次普通的转账交易会需要21 000个Gas，诸如此类，在以太坊中只要是会消耗计算资源的步骤都有个标价。

站在技术和经济的角度来看，通过Gas机制，可以鼓励大家编写更为紧凑高效的合约，避免死循环计算的执行步骤，根据Gas单价设置打包优先级顺序等，这是一种自动化的约束机制，以太坊作为一种公有区块链系统，在去中心自治的前提下，通过一个简单的Gas，让代码的执行具备了成本，从而使得以太坊网络不再是一个简单的软件网络系统，而且是

一个具有金融管控能力的系统。

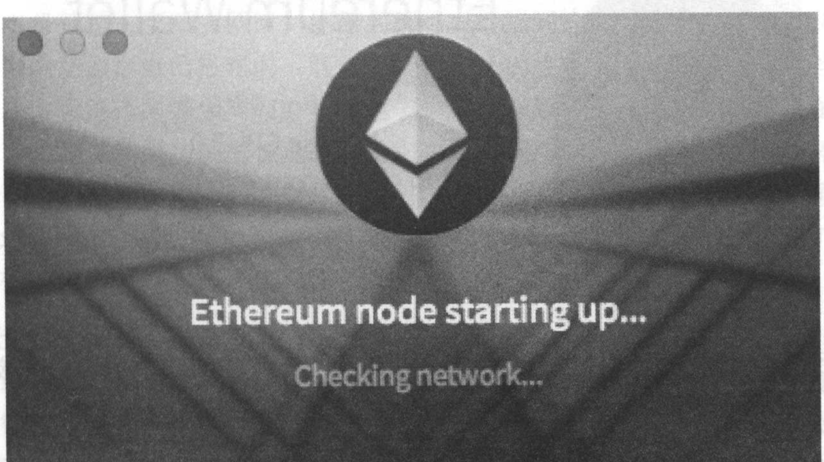
注意　Gas 并不等于以太币，这里有个公式需要说明一下，以太币总额 = 消耗的 Gas×Gas 单价，Gas 单价是可以自己设置的，以太坊客户端一般会设置一个默认的 Gas 单价（0.05e12 wei）以方便使用。在有些操作过程中，比如通过以太币来购买某个投资代币，为了抢夺到优先的打包顺序，往往会设置一个较高的 Gas 单价。当某个账户在发起一个合约操作时，如果执行过程需要的 Gas 大于账户的余额，则执行过程会被中断回滚。

6.1.4　官方钱包使用

钱包客户端是以太坊中一个很重要的面向用户使用的工具，类似于比特币中的钱包功能。作为钱包，其支持的基本功能自然是以太币的转账交易，常见的以太坊钱包有 Ethereum Wallet（这是官方提供的），还有 Parity 以及 imtoken 等，由于以太坊中可以通过智能合约创建自定义的代币，因此通过以太坊钱包除了可以管理自己的以太币外，通常还可以用来管理自己的其他自定义代币资产。在本小节中，我们就来认识一下 Ethereum Wallet，这个钱包客户端是由官方提供支持的，拥有漂亮的图形界面，除了可以用来管理自己的货币资产外，还可以用来部署智能合约，下载地址为 https://github.com/ethereum/mist/releases，大家可以根据自己的计算机操作系统下载对应的版本，下载完毕后可以直接运行安装，无论是哪个操作系统版本，界面和功能都是一致的，我们来看一下 Ethereum Wallet 运行后是什么样子。

1. 启动

我们可以看到启动界面上有一行提示：“Ethereum node starting up...”



意思就是启动一个以太坊节点，但这不是一个钱包客户端吗，怎么还启动一个以太坊

节点？这是因为在安装 Ethereum Wallet 时，同时会安装一个 geth（以太坊节点程序），比如在 Mac OS 系统上，会安装一个 geth 到 /usr/local/bin 目录中，Wallet 需要连接到一个以太坊节点才能工作，连接哪个节点呢？就是连接这个本地的 geth 运行后的节点，至于这个运行节点连接主网络、测试网络还是私有网络，则根据不同的配置有不同的选择。

我们看下 Wallet 与 geth 节点的关系：



可以看到，Wallet 只是一个前端软件工具，真正的幕后英雄是那些以太坊节点组成的网络，那么，话又说回来了，是不是钱包软件都必须要自带一个 geth 运行节点呢？当然不是，对于一些移动钱包、浏览器钱包等类型，用户并不需要在自己的设备上运行一个以太坊节点，而可以选择连接到运行在云端的节点或者一些由服务商提供的节点。

2. 查看版本

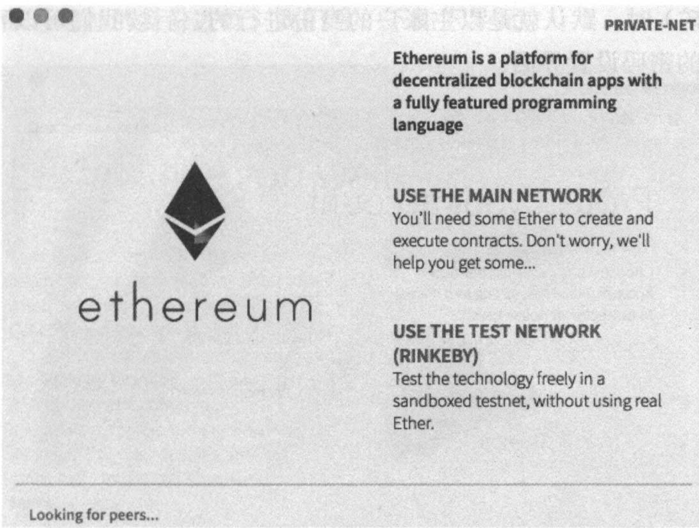
Ethereum Wallet 会不断升级进化，不同的版本在功能和使用上都会有些差异，因此大家在下载使用时，注意查看下自己的版本，这里使用的是 0.9.0 版本，通过菜单栏中的“关于”选项可以查看版本说明。



可以看到，界面中不但可以查看到版本号，还可以看到 GitHub 中的源码链接地址。

3. 网络选择

上面说到了 Wallet 连接 geth 的节点，geth 可以配置成连接不同的网络（主网络、测试网络、私有网络），那么在首次启动 Ethereum Wallet 时，会提供配置选项，如下所示：

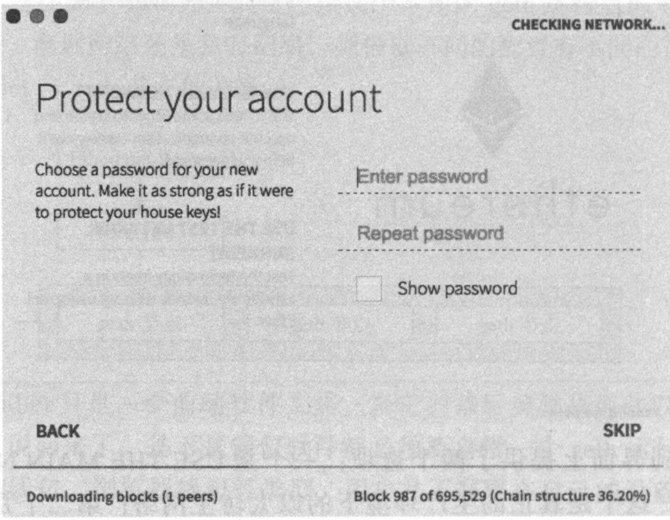


我们可以看到界面上提供了两个选项：一个是 USE THE MAIN NETWORK，也就是主网络的意思，这个是真正的生产环境下的以太坊主网络；第二个是 USE THE TEST NETWORK，也就是测试网络的意思，这是提供给大家用来学习试用的，任何在测试网络上进行的操作都是试验学习用的。因此大家在使用 Ethereum Wallet 时一定要注意当前所处的网络，不要误操作了，我们看到测试网络有个字样叫"RINKEBY"，这是因为以太坊支持多种公共的测试网络，RINKEBY 是目前最新建立的。除了这两个选项，我们还能看到右上角有个字样"PRIVATE NETWORK"，这是私有网络的意思，私有网络就是用户自己搭建的以太网络，如果一个 geth 既没有连接主网络也没有连接测试网络，那就是处于私有网络，有读者可能会有疑问，测试网络是提供测试使用的，那自己搭建的私有网络是不是也能拿来当测试网络？从技术角度来说是这样的，只不过专门的测试网络是以太坊官方启动设立的，其本身就是一个公有网络，任何人都可以连接到这个测试网络，方便大家测试学习，而私有网络是用户自己搭建的，既可以当作自己的生产环境使用，也可以当作自己的测试环境使用。无论选择哪个网络，其功能逻辑都是一致的，并不会因为连入的是主网络功能就多点，是测试网络功能就少点，其差别主要是网络号以及一些运行参数不同，值得说明一点的是，对于 RINKEBY 这个公共的测试网络，其共识算法与主链也不同，使用的是一种叫 PoA(Proof-of-Authority，权威证明）的算法机制，区块由若干个权威节点来生成，其他节点无权生成，从而也就不再需要挖矿了，这主要是为了方便测试使用。

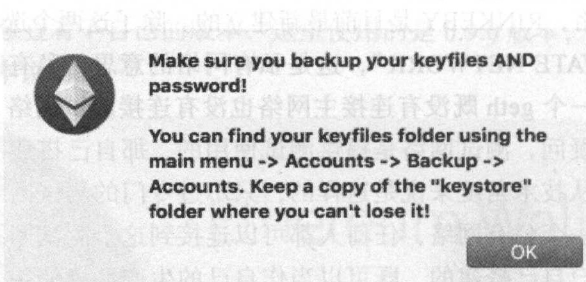
4. 主账户密码设置

选择网络之后，下一步就是设置主账户的密码了，钱包要用来保存用户地址的密钥信息，不设置密码肯定是不行的，那就等于"我家大门常打开，贼儿没事来溜溜"。注意这个密码是我们用来保护默认创建的主账户的，在一个钱包中可以维护多个账户地址，通常第一个创建的账户地址会被作为主账户地址，不过这个是可以更改的。当在钱包中进行各项

功能操作（如挖矿）时，默认就是以主账户的身份进行的，后续我们可以看到相关的操作，先来看下主账户的密码设置界面：



按照界面所示填写密码即可，注意要填写格式复杂一些的密码（比如使用数字字母组合且位数至少6位等），过于简单的密码会校验通不过，填写完后点击进入下一步，会看到弹出一个提醒对话框，显示如下内容：

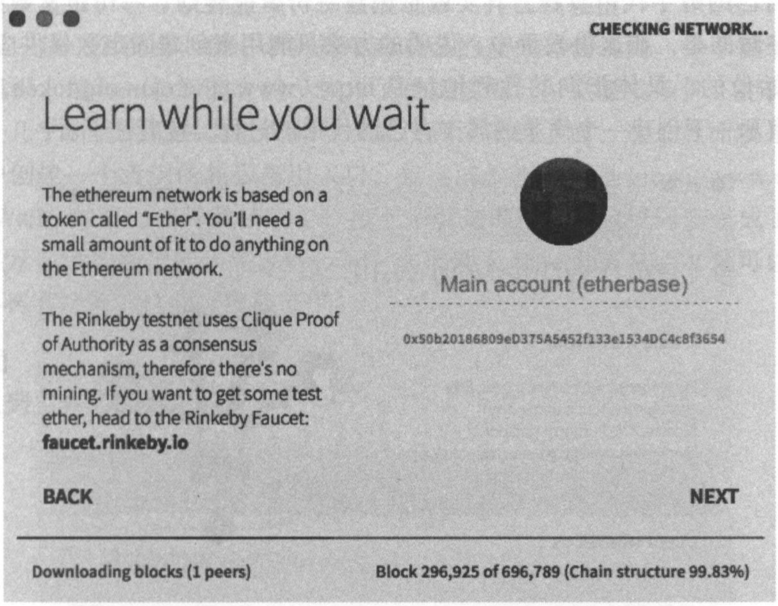


这是提醒用户备份好自己的用户密钥文件，根据界面提示，可以通过菜单栏的"账户"→"备份"→"账户"进入到一个keystore目录，整体备份这个目录即可，如果是创建在主网络中使用的账号，这个目录可千万不能丢失，这比忘记银行卡密码还麻烦，一旦丢失遗忘，找回的机会很渺茫，就可能会遭受惨重的资产损失！

5. 等待时学习

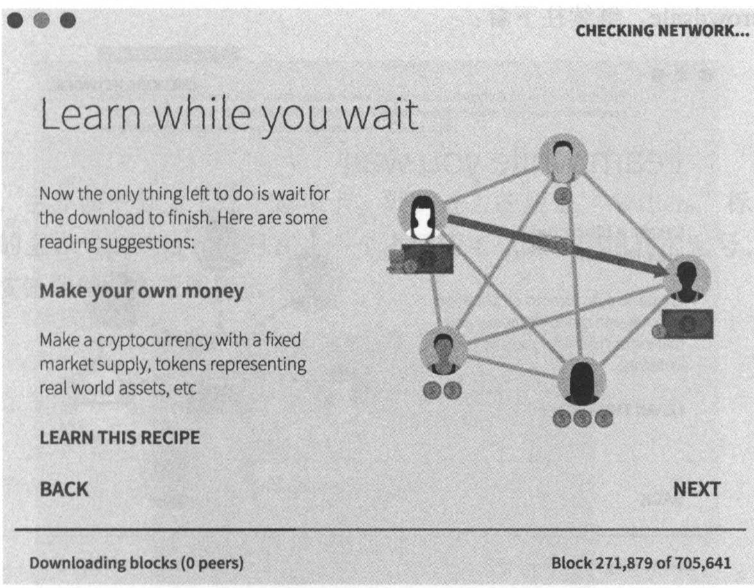
设置完密码，进入下一步后，可以看到提示"Learn while you wait"，意思就是在等待时学习一些知识内容，等待什么呢？等待区块数据的同步，本机的区块链账本数据要与网络中的其他节点同步，我们在界面下方可以看到有进度显示，学些什么？就是了解一些知识片段。同时，我们可以看到界面上显示了一个账号地址，这个就是主账号，可以理解为钱包中的默认使用账号地址，当钱包中有很多个账户地址时，进行挖矿或者合约部署等操

作就会默认以主账号地址的身份进行，我们看一下如下界面：



通过界面上的文字信息，我们可以了解到当前进入的测试网络名称叫 Rinkeby，在这个测试网络中，使用的不是通常的工作量证明共识算法，而是 Clique Proof of Authority（授权证明），因此不需要在 Rinkeby 测试网络中进行挖矿。如果需要获得一些测试使用的以太币，可以访问 faucet.rinkeby.io 获得。

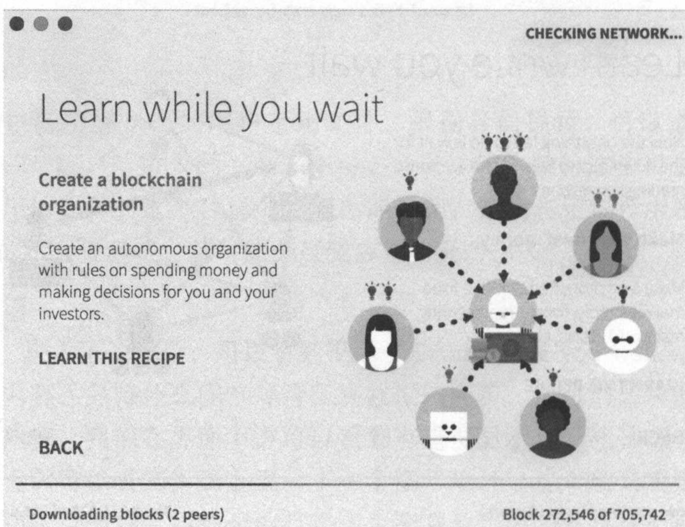
我们接着往下看，下一步：



注意看界面中的标题链接“Make your own money”，这是一个教程的网址链接，告诉你如何创建自己的数字代币教程，其实就是创建一份智能合约，一份定义数字代币的智能合约。值得注意的是，在这份教程中，说明的方法只能用来创建固定数量供应的代币系统，是有一个初始值的，具体指向的教程地址是 https://www.ethereum.org/token。在后续章节中，我们会演示一下创建一个简单的属于自己的代币的过程，接着往下看：



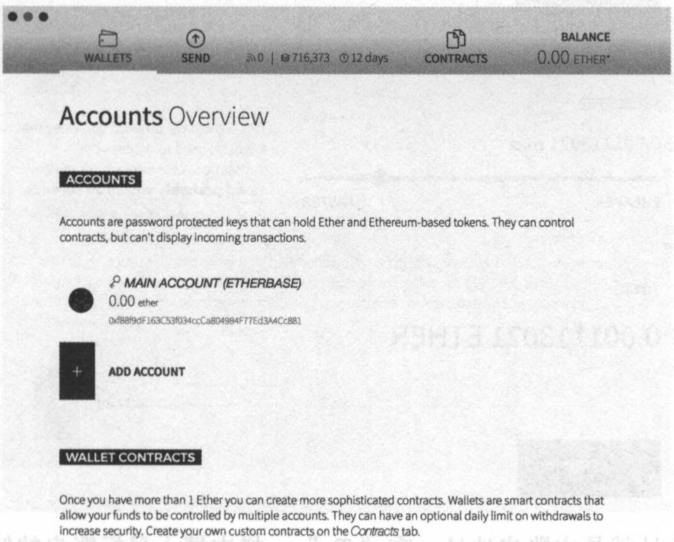
这仍是一个教程的指引界面，刚刚是创建数字代币，这个是创建众筹合约，通过以太坊可以创建一个不需要第三方监管的可信任的众筹合约，由此我们也能看到，以太坊中可做的事情可真是不少，已然不仅仅是数字货币的能力了，其指向的教程地址是 https://www.ethereum.org/crowdsale。继续往下看：



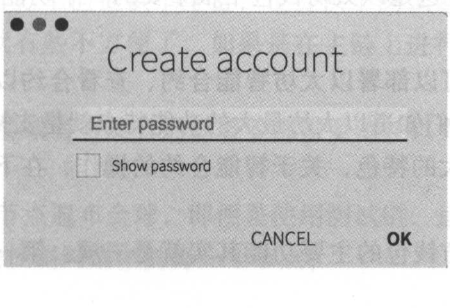
界面中的文字“Create a blockchain organization”是说可以创建一个组织，这是什么意思？刚刚是数字货币，还有众筹合约，那还能理解，什么叫创建一个组织。我们知道，组织与公司一样，是一个机构，在一个机构中有自己的业务运营规则。举个例子，创建一个融资租赁的组织，编写一份智能合约，包含了租赁规则、付款触发条件、担保条件等，我们可以看到，几乎所有的金融类组织都可以架设到区块链上面，这个界面就是一个关于如何在以太坊中创建一个组织的教程指引入口，指向的教程地址是 https://www.ethereum.org/dao。至此，Wallet 的界面引导就结束了，接下来就是等待区块数据同步完成了，在界面的底部有进度显示，区块数据同步完成后，下一次再进入 Ethereum Wallet 就可以直接进入到主界面，而不再有之前的这些过程环节了。

6. 主界面

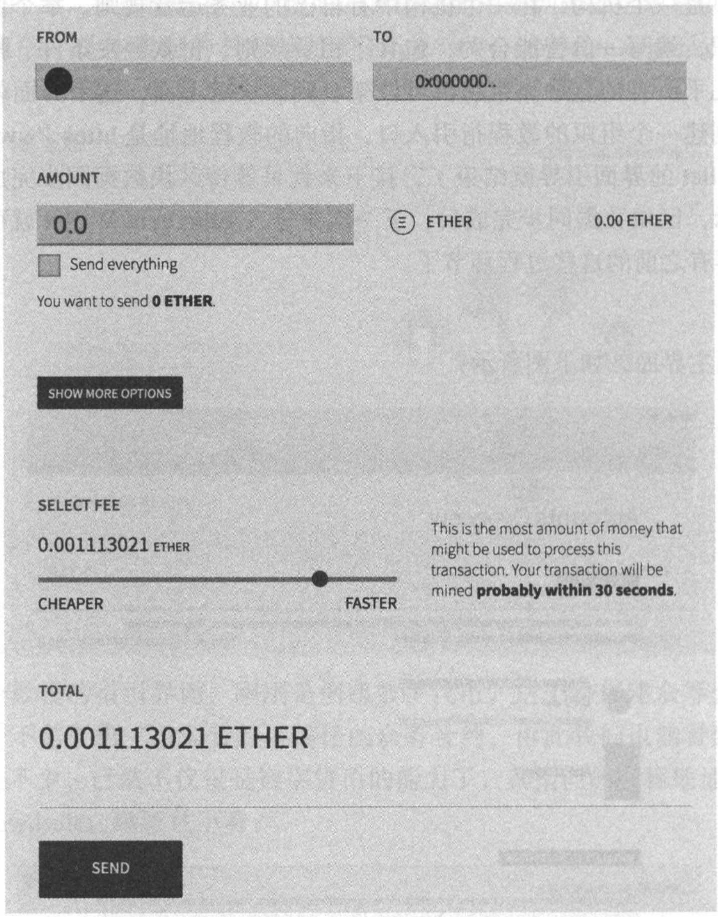
我们进入到主界面，如下图所示：



可以看到，主界面中分为了几个选项卡，默认进去的便是“Wallets”标签页面，可以看到已经创建的主账号的地址，同时在这一个页面还可以继续创建新的账号，点击“ADD ACCOUNT”按钮即可，如下图所示：



只要输入密码即可，操作相当简单。如果要进行转账操作，可以进入第二个标签页“SEND”，如下图所示：



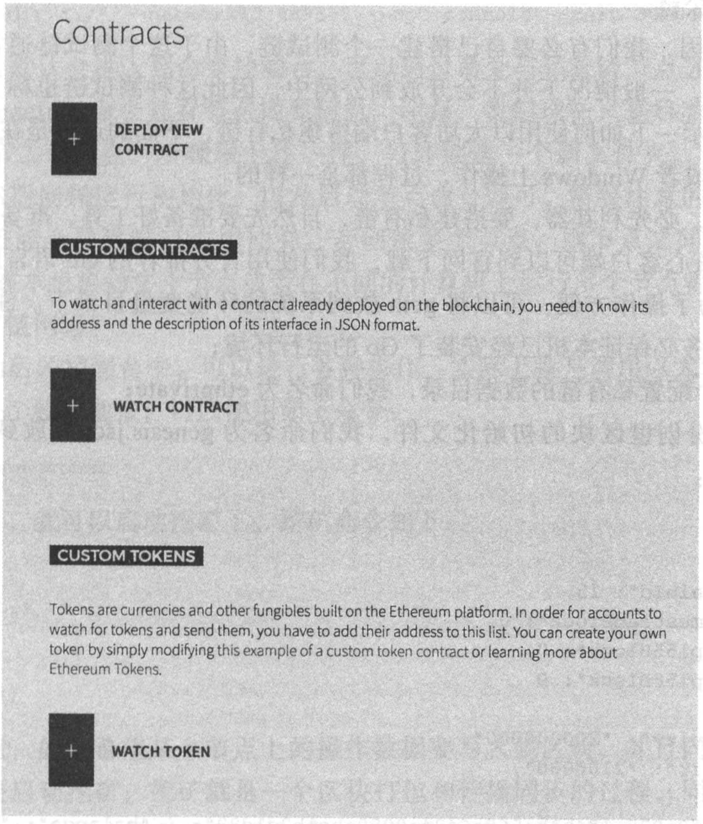
默认的发送地址就是主账户地址，在“To”一栏中填入目标账户的钱包地址，然后在“AMOUNT”中填写转账金额，接着可以选择手续费，如果希望交易事务能被更快地打包进区块，可以选择更高一些的手续费，矿工会优先处理手续费更高的交易事务，都填写完毕后，点击“SEND”即可。

接下来看一个标签页，这是以太坊钱包中很重要的一个功能，那就是智能合约的管理，界面如下图所示。

通过这个页面，我们可以部署以太坊智能合约、查看合约以及查看建立在以太坊之上的数字代币（TOKEN），我们知道以太坊最大的功能特点就是支持智能合约，因此这一块的功能操作是以太坊钱包最大的特色，关于智能合约的操作，在下面章节中有具体的过程演示，这里就不再赘述了。

我们可以发现，以太坊钱包的主要功能其实就是三项：第一是账户地址的管理；第二

是转账交易的操作；第三是智能合约的管理。当然，我们这里说的是以太坊的官方钱包，事实上除了官方钱包，还有一些其他的以太坊钱包软件，有些仅仅提供了账户地址和转账交易的操作功能而省去了智能合约的部署功能，对于一般用户来说，基本也足够了。



6.2　以太坊应用

6.2.1　测试链与私链

以太坊属于公有链，官方不但提供了主链，也提供了测试链，然而对于想要更进一步理解以太坊结构的读者，就有些不方便了，如果是在主链上进行操作使用，则有如下一些问题：

1）以太坊上的转账交易或者智能合约部署等都需要消耗以太币，显然不适合开发测试的需求；

2）以太坊公链的运行节点遍布全球，即便是使用测试链，运行速度也是无法达到实验级的要求的，而且不方便去控制网络中的每一个节点；

3）对于公链的使用，只是通过客户端直接去连接使用，但看不到网络具体是怎么搭建起来的，很多细节看不到；

4）若在某些场合下只是希望使用以太坊来搭建一个局部的网络，类似于局域网，那肯定不能直接使用公链。

基于以上原因，我们有必要自己搭建一个测试链，由于这个测试链通常运行在用户自己的局域网络中，一般情况下并不会开放到公网中，因此这种测试链也称为私有链，在本节，我们就来演示一下如何使用以太坊客户端搭建私有链，下面的过程是在 Mac OS 上完成的，若在 Linux 或者 Windows 上操作，过程都是一样的。

工欲善其事，必先利其器，要搭建私有链，自然先要准备好工具，准备材料如下：

1）以太坊核心客户端可以到官网下载，我们使用官方推荐的 Go 语言版本 geth，下载版本为 1.6.5，为了操作方便，可以将 geth 放到系统的环境变量目录下。注意，由于是 Go 语言版本，因此务必保证本机已经安装了 Go 的运行环境；

2）创建一个配置私有链的数据目录，我们命名为 ethprivate；

3）准备一份创世区块的初始化文件，我们命名为 genesis.json，放到 ethprivate 目录中，其内容如下：

```
{
"config": {
        "chainId": 15,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
    "difficulty": "200000000",
    "gasLimit": "2100000",
    "alloc": {
        "7df9a875a174b3bc565e6424a0050ebc1b2d1d82": { "balance": "300000" },
        "f41c74c9ae680c1aa78f42e5647a62f353b7bdde": { "balance": "400000" }
    }
}
```

可以看到，主要配置了初始的难度值以及两个初始的钱包地址及其余额，这些值大家可以根据自己的需要自行设置。这里需要注意，如果使用的 geth 客户端版本不是 1.6.x 而是 1.5.x（如 1.5.9），则配置文件中不能有 config 段，否则会出错。

接下来创建私有链了，创建的过程相当简单，进入到 ethprivate 目录中，执行如下命令：

```
geth --datadir "./" init genesis.json
```

命令中，通过 datadir 参数指定了数据目录，这里指定的是当前所在的目录也就是 ethprivate 目录，执行完命令后，可以在 ethprivate 目录下看到生成了两个子目录：一个是 geth；一个是 keystore。创建完毕后，我们就可以启动这个私有链了，命令如下：

```
geth --datadir "./" --networkid 989898 --rpc console --port 0
```

可以看到这里指定了 networkid，并且开启了 rpc 服务，当然参数还不止这些，更多的参数应用可以查看 geth 命令的使用帮助。如果要指定端口，可以如下执行启动命令：

```
geth --datadir "./" --networkid 989898 --rpc console --port 30304 --rpcport 8546
```

启动成功后默认会进入到命令控制台，可以通过命令与私有链节点进行访问，我们可以通过 admin.nodeInfo 命令查看节点摘要信息。注意，我们现在只是启动了一个节点，如果还需要启动第二个节点，步骤跟上述一样，另外创建一个新文件夹，将 genesis.json 复制到目录中，然后同样运行初始化，以及启动节点命令即可，需要注意的是，要指定不同的端口，否则可能会导致端口占用冲突。若创建多个节点，则节点之间可以通过 admin.addPeer 连接，在本机启动多个节点或者在不同的计算机上运行多个节点都可以，这样可以模拟出一个私有链网络。

在节点启动后的控制台中，可以进行各种操作，实际上就是调用以太坊节点的 RPC 服务，比如新建一个账户地址，可以使用如下命令：

```
personal.newAccount
```

创建账号后，就可以启动挖矿了，挖矿命令如下：

```
//启动挖矿
miner.start()

//停止挖矿
miner.stop()
```

一定要注意，如果希望某个节点上的操作数据要写入到区块，并且同步到网络中的其他节点，必须要启动挖矿，挖矿就是一个区块打包和传播同步的过程，同时也只有启动挖矿，才能让私有链中的主账号获得以太币，进而可以用来继续测试转账交易、合约部署、合约调用等功能。

有读者可能会觉得这种配置私有链的方式还是有些麻烦。对于想快速方便进行测试使用以及智能合约开发的读者，还有一种配置私有链的方式，那就是使用 TestRPC 与 Truffle 组合。TestRPC 是在本地使用内存模拟的一个以太坊环境，可以用于搭建测试环境，基于 Node.js 开发，因此使用 TestRPC 首先要安装 Node.js 环境并且版本要大于 6.9.1。Truffle 是针对以太坊智能合约应用的一套开发框架。我们来看下具体如何搭建。

1）使用 npm 安装 TestRPC，命令如下：

```
sudo npm install -g ethereumjs-testrpc
```

2）安装完毕后，可以输入如下命令查看版本信息：

```
testrpc -version
```

3）运行后输出如下信息：

```
EthereumJS TestRPC v3.0.5

Available Accounts
==================
(0) 0xb43333d44136f351fd30d20215490432e0f3968d
(1) 0x34f73354540fa4b653bf33568c7ba9f69ad6c84d
(2) 0x047bf66a5be28bb84502f3baaa40b10cb04d44e8
(3) 0x1dc4d3ce33b05e24219f93f28612b9a80e2724de
(4) 0xaa82bb04532d560139eeae495fc6d00706dbc7f7
(5) 0xe2ee5d9e955277b6f5e13d10beb686e069859731
(6) 0xfe023592bc7bbb7dac6051950a0b8774206c1f5b
(7) 0x34b1b1b6a36348912be0b943e3f34db38339a192
(8) 0x860638afa0bbecf8ea5c5808e95108710ec92acc
(9) 0xd7a701bd9cffbf887b1e3f03ac91c68ead3032ec

Private Keys
==================
(0) 6e94ed2e32818d2bb1d58bd0119407096691ec683ed8a43a2975ff6003bb1924
(1) ccc8fca886b7666c0e2707cc9a429d4a1c941dd170b8c0f5f55c71ce966fa835
(2) d0ba307ed8ff2ec12deeb59d0c85884d74735b8ab26329e74cb37966f656634b
(3) c057235669dd341ebb4ce1185b469eeac3d1cd2f763e95aa36e0e22adaa9ce84
(4) 7c6d6a7268fd9f76d2868f650168ba67a2901d427de85357b6a453dfad784db4
(5) 2baaacc4818dfc605b0c91ba5418826a86c09b375861123aa393d7411cce6595
(6) 147d63765df501f9417951445966050 2b00692b4aab0dfcc2316aea9fc0877dd
(7) bbded13290bcefc551d1cc9bf36921a2e65024b03401014388a6ce52c2159494
(8) 53df9b7d172746d9a0a548f62bc1d21ca78bd6202456221241b36a7fa1cf3b91
(9) 03691c7d31ce9b041d2a66bdc48397b13660f097164d443e3f8ba3f09ae9272e

HD Wallet
==================
Mnemonic:      trade target identify fun bleak wish sphere emotion journey rose decide above
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

可以看到，默认就自动配置了 10 个账户地址。注意，以上信息是动态的，每次启动时随机生成，不是固定的。通过最后一行数据的提示，表明 TestRPC 启动后使用 8545 端口监听。

4）同样使用 npm 安装 Truffle，命令如下：

```
sudo npm install -g truffle
```

5）安装成功后，输入 truffle --version，输出如下：

```
Truffle v3.2.5 - a development framework for Ethereum

Usage: truffle <command> [options]

命令:
  init      Initialize new Ethereum project with example contracts and tests
  compile   Compile contract source files
  migrate   Run migrations to deploy contracts
  deploy    (alias for migrate)
  build     Execute build pipeline (if configuration present)
  test      Run Mocha and Solidity tests
  console   Run a console with contract abstractions and commands available
  create    Helper to create new contracts, migrations and tests
  install   Install a package from the Ethereum Package Registry
  publish   Publish a package to the Ethereum Package Registry
  networks  Show addresses for deployed contracts on each network
  watch     Watch filesystem for changes and rebuild the project automatically
  serve     Serve the build directory on localhost and watch for changes
  exec      Execute a JS module within this Truffle environment
  version   Show version number and exit

See more at http://truffleframework.com/docs
```

可以看到，输出了版本信息 v3.2.5，并且在下面列出了各种可以使用的命令。

6）安装 solc：

```
sudo npm install -g solc
```

注意，安装后的命令是 solcjs，这是用来编译智能合约代码的。

7）运行测试

首先运行 TestRPC，在命令行中直接通过 testrpc 命令可以启动，接着开始初始化 Truffle 目录，命令如下：

```
mkdir mytruffle && cd mytruffle
truffle init webpack
```

这个命令其实就是下载一个项目框架，也可以直接通过网址 https://github.com/trufflesuite/truffle-init-webpack 下载压缩包后解压缩，复制到相应目录，效果是一样的。初始化命令运行后，输出如下信息：

```
Downloading project...
Installing dependencies...
Project initialized.

  Documentation: https://github.com/trufflesuite/truffle-init-webpack

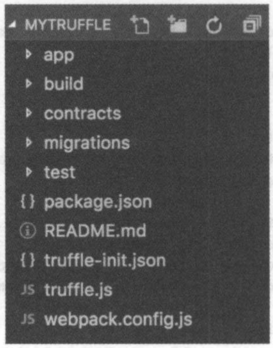
Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test:           truffle test
  Build Frontend: npm run build
  Run Linter:     npm run lint
  Run Dev Server: npm run dev

Hint: Run the dev server via `npm run dev` to have your changes rebuilt automatically.

Make sure you have an Ethereum client like the ethereumjs-testrpc running on http://localhost:8545.
```

提示信息很简单，就是下载项目框架后进行初始化，初始化完成后，在目录中生成了如下文件：



其中，contracts 中存放的是合约，truffle compile 进行编译的时候就会在这里面寻找合约文件，migrations 目录里面存放的是 JavaScript 文件，它帮助部署合约到以太坊网络中，它们表示了进行部署任务的步骤，truffle.js 文件内容如下：

```
module.exports = {
  networks: {
    development: {
      host: "localhost", // 节点地址，如果是私有链，一般是本机
      port: 8545,   // 节点 RPC 端口
      network_id: "*" // 自定义网络号
    }
  }
};
```

默认的配置与 testrpc 的参数是一致的，也可以根据需要修改。

8）启动了 testrpc，也初始化了 truffle，现在开始试着编写合约。可以看到在 contracts 中已经有几个示例合约了，不用管，创建一个 MyCalc.sol，源码如下：

```
pragma solidity ^0.4.11;
contract MyCalc {
  function SumAdd(uint a) returns(uint d) {
    return a + 100;
  }
}
```

这是一段非常简单的代码，合约名为 MyCalc，其中包含了一个方法 SumAdd，通过传入一个整数参数，返回一个加上 100 的值，这就是一份智能合约，智能合约并没有我们想象得那么复杂，与传统应用软件开发最大的区别就是，编写的合约一旦部署到以太坊上，就会被同步到每一个节点中，由整个以太坊网络的基础设施来确保合约的刚性执行以及不可篡改性。我们先来看下这份编写的合约如何部署执行，代码编写完毕后就可以进行编译了，以太坊中的智能合约都运行在 EVM（Ethereum Virtual Machine，以太坊虚拟机）上，

必须首先被编译为EVM能识别的字节码。

9）回到mytruffle的目录中，进行编译，执行如下命令：

```
sudo truffle compile
```

10）编译若没有问题，则会有如下提示：

```
Compiling ./contracts/MyCalc.sol...
Writing artifacts to ./build/contracts
```

11）可以看到生成了一个build目录，编译没有问题就可以部署了，进入到migrations目录，编辑“2_deploy_contracts”文件，在最后一行插入“deployer.deploy”（合约名），编辑内容如下：

```
var ConvertLib = artifacts.require("./ConvertLib.sol");
var MetaCoin = artifacts.require("./MetaCoin.sol");
var MyCalc=artifacts.require("./MyCalc.sol"); // 新增

    module.exports = function(deployer) {
     deployer.deploy(ConvertLib);
     deployer.link(ConvertLib, MetaCoin);
     deployer.deploy(MetaCoin);
     deployer.deploy(MyCalc); // 新增
   };
```

12）编辑保存后，执行部署命令：

```
sudo truffle migrate
```

13）注意，在操作过程中一定要保证TestRPC是开启的，命令执行成功后，在TestRPC中可以看到响应，接下来调用一下合约中的方法，要调用合约的功能，得与TestRPC模拟节点交互，首先进入到控制台，命令如下：

```
sudo truffle console
```

14）进入到控制台后，我们进入到MyTruffle目录下的build子目录中，找到MyCalc.json，打开它找到abi的内容段，复制出来，然后回到控制台，执行如下命令：

```
abi= 复制出来的 abi 内容
```

15）同时在MyCalc.json中找到合约的地址，并且在控制台中执行命令：

```
myContract=web3.eth.contract(abi).at("0xc7b8a297b99e473feeaf447993600336482c8a
8a")
```

16）接下来就可以执行合约中的函数了，执行如下命令：

```
myContract.SumAdd.call(10)
```

最后就能看到结果了。

至此，我们对以太坊的私有链配置以及开发测试环境的搭建介绍就结束了，大家可以根据自己的具体需求进行各项参数的配置，另外也要注意版本变更带来的一些问题，比如 geth 的不同版本之间会有些差异，例如 1.6 版本去除了内置的 JavaScript 环境编译功能，而 1.5 版本中是有的，因此在 1.5 版本中可以直接使用相关的合约编程和编译功能。以太坊是一个开源系统，功能开发也一直处于不断的进化中，随着发展，也会出现更方便的功能，大家在具体使用过程中可以多关注一下。

6.2.2 编写一个代币合约

在上面几节中，我们演示了一段合约代码的编写，不过只是一个简单的加法运算，实在让人感觉不到智能合约的特色。在本小节中，我们来演示一下如何通过以太坊智能合约来创建一个数字代币，我们参照以太坊官网的示例，提供一段最简单的代币合约示例，代码如下：

```
pragma solidity ^0.4.11;
contract MyToken {
    //声明数组，用以存储代币所有人的地址列表
    mapping (address => uint256) public balanceOf;

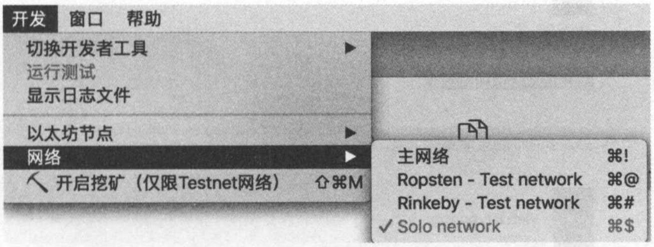
    // 初始化代币总额，赋值给合约创建者的账户地址中
    // 这是一个构造函数，只会被执行一次
    function MyToken(
        uint256 initialSupply
        ) {
        balanceOf[msg.sender] = initialSupply;
    }

    /* 代币发送 */
    function transfer(address _to, uint256 _value) {
        require(balanceOf[msg.sender] >= _value);              // 检查余额是否足够
        // 检查是否会溢出，主要防止循环发送给自己
        require(balanceOf[_to] + _value >= balanceOf[_to]);
        balanceOf[msg.sender] -= _value;        // 从发送者账户中减掉发送的金额
        balanceOf[_to] += _value;               // 在接收者账户中增加发送的金额
    }
}
```

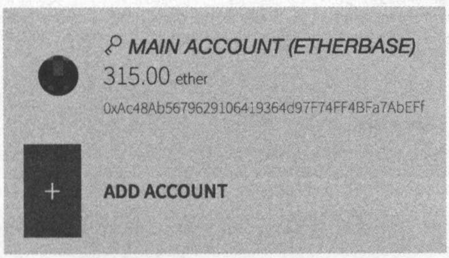
通过上述代码的注释说明，我们大致了解了本合约代码中定义了一个名字叫 MyToken 的代币，提供了一个构造函数初始化代币数量，构造函数中的 msg.sender 是指当前调用者的以太坊账户地址，由于合约一般都是由创建者部署的，因此初始化的代币会通过执行构造函数一次性全部记录在创建者的账户地址中。除了构造函数，本合约还提供了一个发送的方法，用来进行代币转账，逻辑很简单，参数中包含了一个转账目标账户地址的参数 _to 和一个转账金额参数 _value，过程就是做一些基本校验以及更改转出和转入账户的金额，

我们通过以太坊官方钱包来部署，为了方便操作，我们使用较新的 0.9.0 版，其自带的 geth 是 1.6.5 版，并且支持直接配置为本机单节点私有链，我们来看一下操作步骤。

（1）配置为单节点私有链



可以看到，配置相当简单，选择" Solo network"即可，选择后可以创建一个账户，然后点击"开启挖矿（仅限 Testnet）网络"，可以看到创建的账户中很快就能获得以太币，如下所示：



由于部署合约以及调用合约方法要消耗以太币，因此读者可以根据自己的需要确保账户中具备足够的余额。

（2）代币合约部署

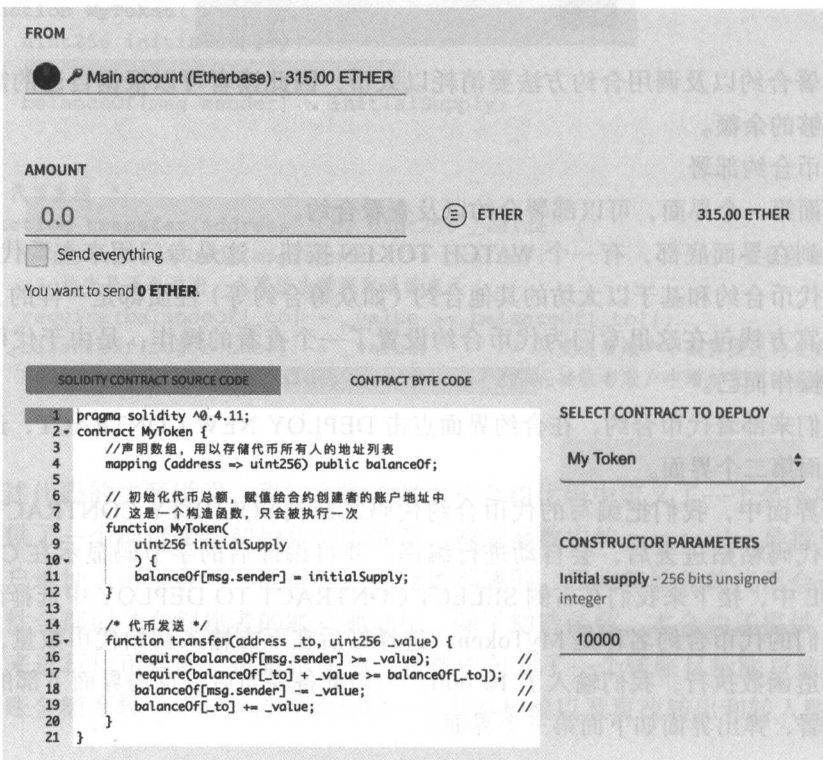
通过下面第一个界面，可以部署合约以及查看合约。

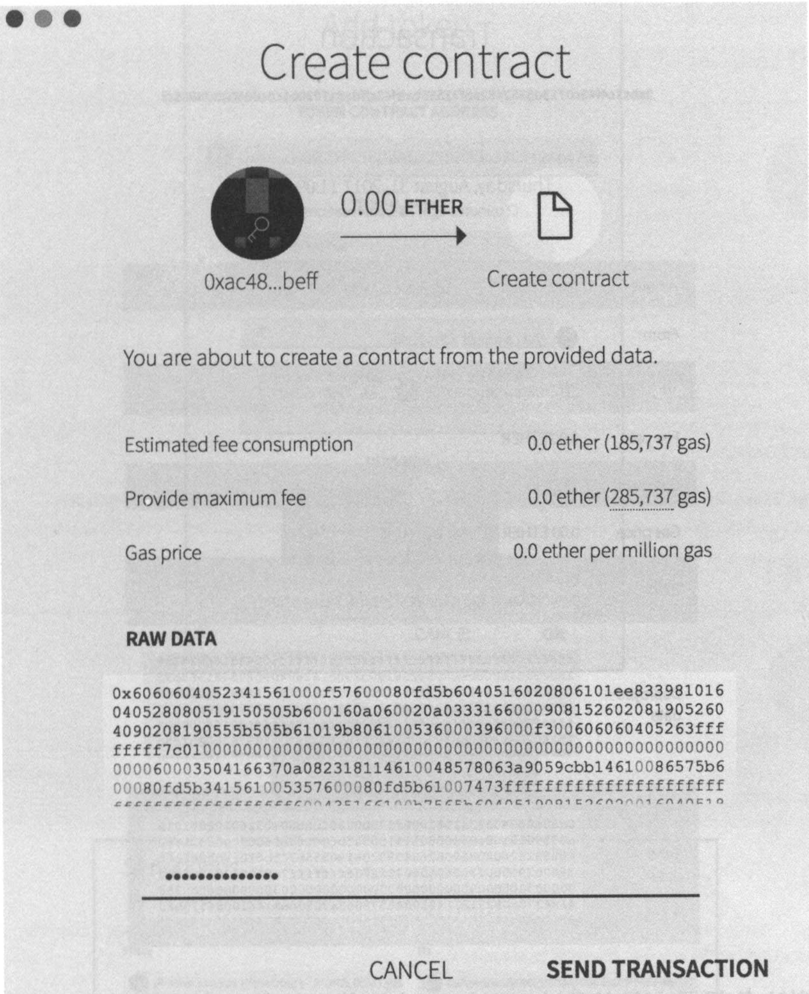
我们看到在界面底部，有一个 WATCH TOKEN 按钮，这是专门用来查看代币合约的，大家注意，代币合约和基于以太坊的其他合约（如众筹合约等）性质都是一样的，只是代码逻辑不同，官方钱包在这里专门为代币合约设置了一个查看的操作，是由于代币合约的特殊性，方便操作而已。

现在我们来部署代币合约，在合约界面点击 DEPLOY NEW CONTRACT，进入到部署界面，如下面第二个界面。

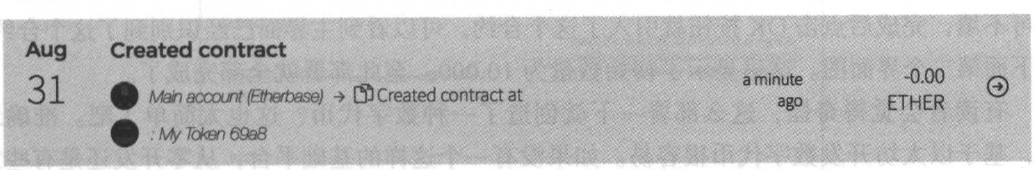
在这个界面中，我们把编写的代币合约代码复制到 SOLIDITY CONTRACT SOURCE CODE 中，代码粘贴进去后，会自动进行编译，并将编译后的字节码显示在 CONTRACT BYTE CODE 中，接下来我们在右侧 SELECT CONTRACT TO DEPLOY 中选择待部署的合约名称，我们的代币合约名称是 MyToken，选择好后在下方输入初始代币数量，这个数量会提供给构造函数执行，我们输入了 10 000。一切准备妥当后，点击界面下部的 DEPLOY 按钮执行部署，弹出界面如下面第三个界面。


PRIVATE-NET
0 | 63
an hour
WALLETS
SEND
CONTRACTS
BALANCE
315.00 ETHER*
Contracts
DEPLOY NEW CONTRACT
CUSTOM CONTRACTS
To watch and interact with a contract already deployed on the blockchain, you need to know its address and the description of its interface in JSON format.
WATCH CONTRACT
CUSTOM TOKENS
Tokens are currencies and other fungibles built on the Ethereum platform. In order for accounts to watch for tokens and send them, you have to add their address to this list. You can create your own token by simply modifying this example of a custom token contract or learning more about Ethereum Tokens.
WATCH TOKEN

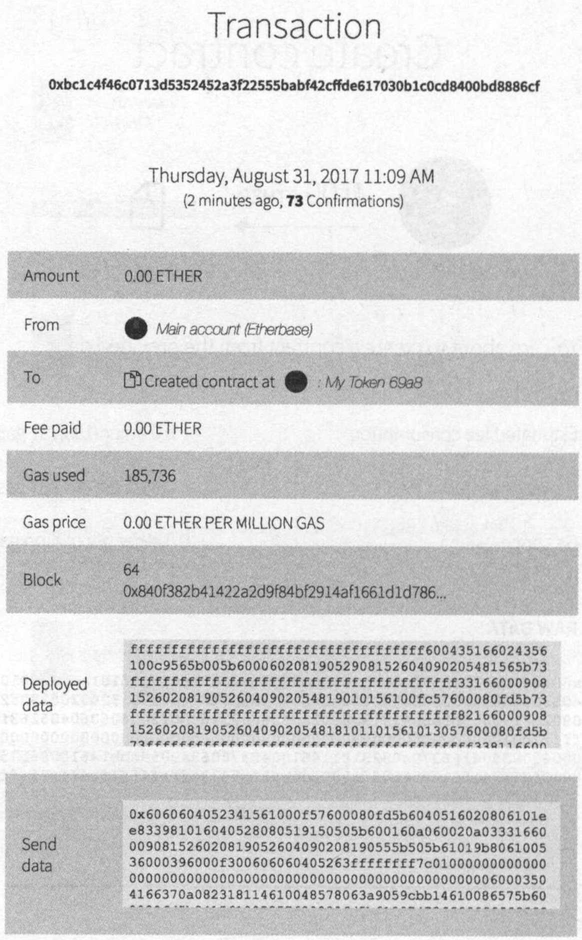

FROM
Main account (Etherbase) - 315.00 ETHER
AMOUNT
0.0
ETHER
315.00 ETHER
Send everything
You want to send 0 ETHER.
SOLIDITY CONTRACT SOURCE CODE
CONTRACT BYTE CODE
pragma solidity ^0.4.11;
contract MyToken {
//声明数组，用以存储代币所有人的地址列表
mapping (address => uint256) public balanceOf;
// 初始化代币总额，赋值给合约创建者的账户地址中
// 这是一个构造函数，只会被执行一次
function MyToken(
uint256 initialSupply
) {
balanceOf[msg.sender] = initialSupply;
}
/* 代币发送 */
function transfer(address _to, uint256 _value) {
require(balanceOf[msg.sender] >= _value);
require(balanceOf[_to] + _value >= balanceOf[_to]);
balanceOf[msg.sender] -= _value;
balanceOf[_to] += _value;
}
}
SELECT CONTRACT TO DEPLOY
My Token
CONSTRUCTOR PARAMETERS
Initial supply - 256 bits unsigned integer
10000



上图显示了合约创建的一些摘要信息，部署的账户地址以及需要耗费的 Gas，在主链上部署时会根据 Gas 以及 Gas Price 计算出需要花费的以太币金额，我们现在是在单机私链上操作，因此没有这些限制，可以直接进行部署，点击 SEND TRANSACTION 即可发起一个部署交易，执行后回到合约主界面，可以看到底部显示部署状态，如下所示：



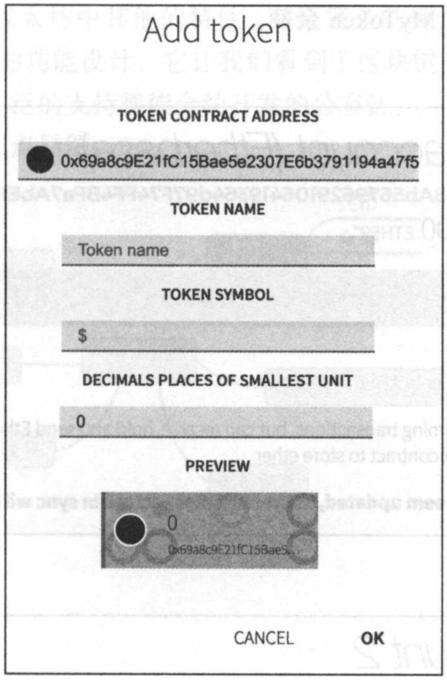
读者可能发现一直显示着这个状态，似乎部署不上去，原因可能是没有开启挖矿，部署的时候必须处于挖矿状态才能成功，开启挖矿后就部署成功了，部署后可以查看这条部署交易信息：



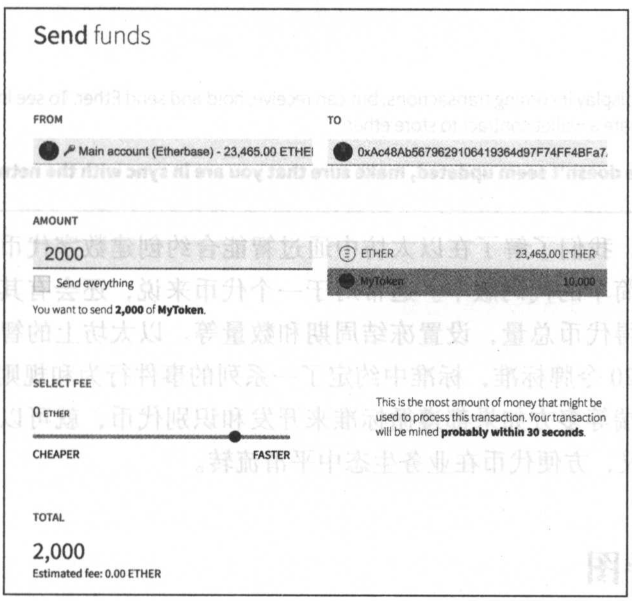
可以看到这条部署交易的账户地址、已确认的区块数、合约部署的区块高度等信息，也可以看到部署的合约地址。在以太坊中，部署合约也属于一种交易事务。部署完成后，我们可以在合约操作的主界面上点开 WATCH TOKEN，让以太坊钱包以数字货币的视角来识别这份智能合约，打开后界面如下面第一个界面图。

在 TOKEN CONTRACT ADDRESS 中输入合约地址即可，下面是名称、符合等信息，可填可不填，完成后点击 OK 按钮就引入了这个合约，可以看到主界面已经识别到了这个合约。如下面第二个界面图。这里显示了初始数量为 10 000，至此部署就全部完成了。

有读者会觉得奇怪，这么部署一下就创造了一种数字代币？这也太简单了吧。准确地说，基于以太坊开发数字代币很容易。如果没有一个这样的基础平台，从零开发还是有些复杂的，实际上，以太币本身也是一种合约，只不过合约规则是固化在以太坊代码中的。部署完成后，就可以发送给账户地址了，我们创建另外一个以太坊账户地址，然后做一次转账操作，进入到 SEND 界面，选择主账户并选择 MyToken 代币，如下面第三个界面图所示。



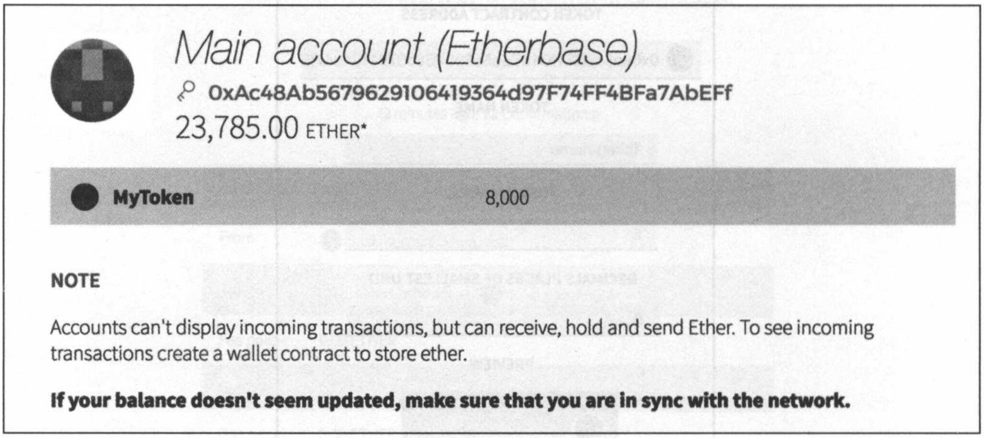




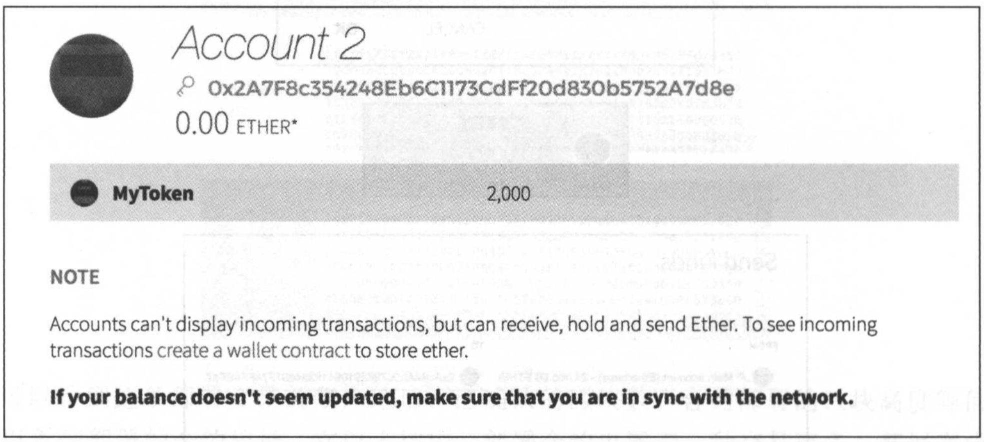
在转账金额一栏填入一个金额（如 2000），填写完毕后点击“发送”即可，我们分别来

看一下主账户和接收账户的 MyToken 余额。

1）主账户：



2）接收账户：



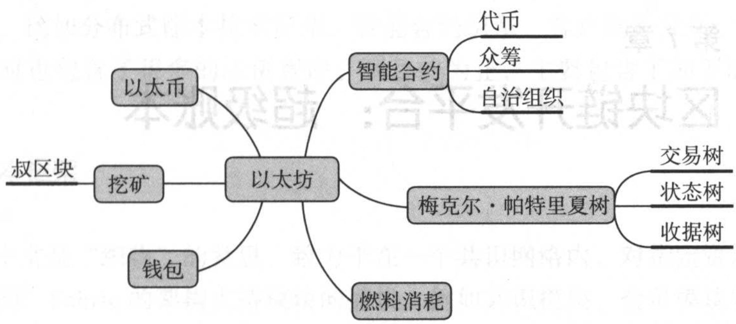
通过上述介绍，我们了解了在以太坊中通过智能合约创建数字代币的过程，当然我们只是演示了一个最简单的代码版本。通常对于一个代币来说，还会有其他多项功能，比如获得账户余额，获得代币总量，设置冻结周期和数量等。以太坊上的智能合约代币有一个标准，也就是 ERC20 令牌标准，标准中约定了一系列的事件行为和规则，应用开发者、交易平台、钱包客户端等多方如果都遵循标准来开发和识别代币，就可以做到事先的接口对应，类似于一个协议，方便代币在业务生态中平滑流转。

6.3 知识点导图

以太坊的出现扩展并丰富了比特币中的脚本思想，使之发扬光大，成为一个通用的智

能合约编程平台。相比于以太坊中其他的特性（如更复杂的梅克尔树、叔区块、燃料消耗等），智能合约是最有价值的功能设计，它让我们看到了区块链技术可以应用的场景，迄今为止，以太坊仍是使用最广泛的支持智能合约开发的公有链。

我们看下以太坊的知识点导图：



Chapter 7 第 7 章

区块链开发平台：超级账本

7.1 项目介绍

7.1.1 项目背景

比特币网络主要的功能就是维持着比特币这种加密数字货币，虽然也能通过扩展开发，但是功能很有限（未来如果比特币源码经过不断的升级，能够完整支持智能合约等更高级技术后会有改观）。以太坊网络维持着以太币同时提供了智能合约的开发和部署，这些合约的运行也都是建立在以太坊的基础之上的。就这两者本身，都只是提供了最基础的基础设施功能，就好像划了一块地，通上了基本的水电和电话，其他所有的建造就都要靠自己了。这对于很多用户来讲建造成本还是大了些，实现自己想要的功能颇多不便，而更关键的是，在很多应用场合并不需要数字货币这个功能，比如：公司内部的账本审计，还有很多场合需要有明确的权限控制，如企业的供应链系统，还有一些场合不适合运行工作量证明这种共识算法，如金融机构之间的支付结算。

除了这些问题，还有一个较大的问题，那就是在一个公链系统上，它的数据在理论上都是不完全确定的，因为在公链环境下只能做到最终一致性（就好像比特币会建议一笔交易数据至少要等待经过 6 个区块的确认才算是比较保险），这对于商业环境下的使用是不能接受的，于是，超级账本应运而生，超级账本实际上是一套开发框架或一组开发资源。

超级账本项目正是由 Linux 基金会主导推广的区块链开源项目，其中汇集了金融、银行、物联网、供应链、制造等各界开发人员的努力支持，其目的是打造一个跨领域的区块链应用。比起比特币、以太坊，超级账本完全就是一个豪门贵族，衔着金钥匙出生的。

7.1.2　项目组成

超级账本项目从创建之初就是一个非常开放的项目组织，由于是面向企业级的服务项目，因此与比特币、以太坊这些公链系统有很大的区别。事实上，超级账本中的项目提供的都是框架级的服务功能，更多的是面向企业级开发的，孵化的项目包含了一系列的企业级区块链技术，比如分布式账本技术框架、智能合约引擎、客户端开发库、图形用户界面、工具库等，同时也包含了很多的示例程序。到目前为止，主要包含了如下的框架项目和工具项目。

1. 超级账本框架

（1）Fabric

Fabric 的中文是"织物"的意思，致力于在一个共识网络内，对指定资产的信息进行互换、维护和调阅。Fabric 的架构支持模块的插拔，例如共识模块、会员模块等。它将进一步推广"智能和约"在容器技术中的应用，从而实现各种商业应用场景。

使用 Fabric 可以开发出比特币这样的应用程序，也可以开发出金融资产交换、账本审计系统等应用，系统中的各个模块（如共识算法）都是可以装配替换的，这个是非常重要的，可以为商业应用提供很灵活的配置。事实上 Fabric 包含着众多的组件模块，比如加密安全、身份鉴权、智能合约、数字资产、可插拔共识算法等。这个项目在超级账本中占据着非常重要的地位，我们所看到的大部分区块链应用，主体功能都可以使用 Fabric 来实现，因此它是一个区块链应用开发的底层设施。目前，全球安全金融信息服务提供商 Swift 已经正式选择在自己最突出的区块链项目中使用超级账本 Fabric 数据库，如果这个区块链概念验证（PoC）获得成功，可以节约高达 30% 与跨境支付相关的和解成本。

（2）Sawtooth

代号"锯齿"，它是又一个企业级区块链账本项目，其主要理念是保持分布式账本的分布式特征，并使智能合约保持安全，这对于企业应用很关键。与 Fabric 一样，Sawtooth 也是高度模块化的，可以根据自己的需要组装不同的功能模块（如共识算法策略）。Sawtooth 支持全新的共识机制 Proof of Elapsed Time（时间消逝证明），这个项目来自 Intel 的代码贡献。

（3）Iroha

本项目的目的是将分布式账本技术便捷地应用于现有的基础项目上，其特点是实施简易、采用了领域驱动 C++ 设计，提供移动应用的开发支持，还支持一种新的拜占庭容错共识算法，名字叫 Sumeragi。这个项目由日本 Sotamitsu 公司提供主要代码贡献。这个项目可以看作对 Fabric 和 Sawtooth 的补充，主要提供移动端的开发。

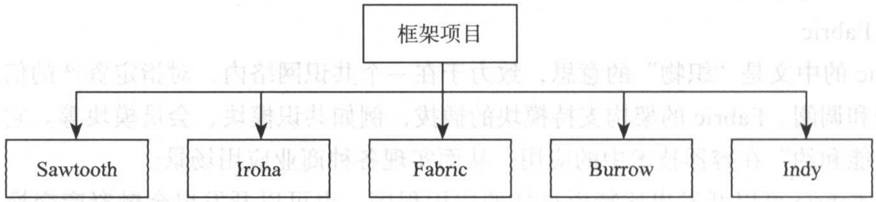
（4）Burrow

这个项目最初是由 Monax 和 Intel 孵化，这是一个授权的智能合约机或者说是一个授权的区块链节点，这个节点可以执行以太坊规范的智能合约代码。从这个角度来说，相当于

以太坊的一个派生项目，Burrow 是被设计为针对多链领域构建的，其主要包含三个组件：共识引擎、以太坊虚拟机以及 rpc 网关。

（5）Indy

这是一个区块链数字身份项目，旨在为区块链生态系统构建数字身份认证工具，这个项目是由 Sovrin 基金会发起的，Sovrin 基金会是为管理世界上第一个自我主权身份（SSI）网络而设立的国际非营利私人组织，这个项目现在也加入了超级账本的阵营。Indy 项目所支持的概念是“可验证的声明”，这是一种加密认证的在线识别理念，私人数据不会被写入账本，哪怕是加密的形式，它与账本绑定，有证据表明它在某个时间是存在的。



2. 超级账本工具

（1）Cello

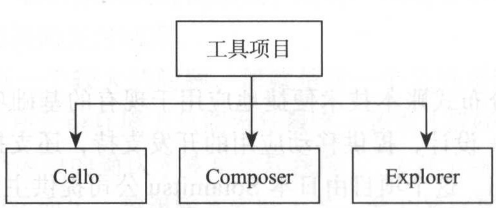
这个工具的主要目的是实现“区块链即服务”（BaaS）的部署模型，类似于“软件即服务”的思想，这种方式提供一个多租户的上链服务。与目前的云服务思想类似，方便区块链应用的生态管理，使用 Fabric、Iroha、Sawtooth 开发的应用都可以通过 Cello 来部署。

（2）Composer

这是一种协作工具，目的是简化和促进超级账本区块链应用，目前 Composer 的所有工作都是在 Fabric 上完成的，不过 Composer 的设计可以支持其他的框架技术，不同的框架支持不同的智能合约的不同实施，通过使用 Composer 可以将这些实施连接在一起。

（3）Explorer

这是一个浏览器工具，可以查看或调用各种区块数据、网络信息、智能合约等，也可以用来部署合约，类似于钱包这个级别的工具。



将来超级账本中的项目可能会越来越多，共同组成一个功能强大且多样的区块链开发资源，正所谓授人以鱼不如授人以渔，立足在技术开发上，提供更多有意义的工具，将极大地推进区块链领域的生态发展。在某种程度上，超级账本已经是属于区块链发展的第三代技术了，在数字货币、金融等领域之外，全面地支持各种场景下的应用开发。

7.2 Fabric 项目

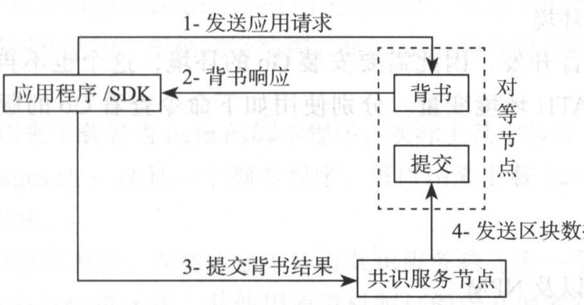
7.2.1 Fabric 基本运行分析

Fabric 项目的目的是要实现一个面向商业环境的通用权限区块链底层设施，它是用于开发企业级区块链应用的主要框架，2017 年 7 月 11 日，官方网站宣布发布了 1.0 正式版，标志着这个框架已经可以进入到生产环境的实践阶段。本节将会通过部署运行一个示例来体会一下如何使用 Fabric 实现一个智能合约系统并且部署一个网络，在开始之前，我们来了解一下 Fabric 与比特币、以太坊这些区块链系统有什么差别，以及它的运行框架是什么样的。

先看一下彼此的区别：

	比特币	以太坊	Fabric
加密数字货币	比特币	以太坊 / 合约代币	不支持
网络权限	完全公开	完全公开 / 许可	许可
交易事务	匿名	匿名 / 私有	公开 / 机密
共识机制	PoW（工作量证明）	PoW（工作量证明）	PBFT（实用拜占庭容错）
智能合约	不支持	支持	支持

从上表可以看出，Fabric 与其他公链系统主要的区别是一个带有许可授权的区块链网络系统，并且不使用通常依靠算力的工作量证明共识算法，而是使用更适合在商业环境下使用的 PBFT 算法。既然 Fabric 有着许多的不同，那么它的运行框架是什么样的呢？我们就看一下 1.0 正式版的运行框架图：



通过上图，我们可以看到对等节点分解为两个角色：一个背书，一个提交。整个运行环路如下：

1）应用程序通过 SDK 调用发送请求到对等节点；

2）对等节点通过智能合约执行请求，请求完毕后会进行背书，背书就是节点对请求执行的确认，返回 YES 或 NO，参与背书的对等节点将执行结果返回给应用程序；

3）应用程序将接收到的背书结果提交给共识服务节点；

4）共识服务节点执行共识过程，生成区块数据并发送给对等节点；

5）对等节点进行交易数据的验证之后再提交到本地的账本数据中。

这便是 Fabric 中的一个交易运行流程，这里提醒一下读者，Fabric 的版本发布过程中，*有过一个重要的版本 0.6 版，这个版本与 1.0 正式版的流程略有差别，这里不再赘述，读者*在试用 Fabric 时，务必注意下版本，接下来我们就开始安装使用 Fabric 示例。

7.2.2 Fabric 安装

要使用 Fabric，首先要安装这个项目框架，本示例在 Mac 上进行，对于 Linux 和 Windows 过程是类似的，不过还是建议使用 Mac 或者 Linux，能省去很多麻烦。对于 Linux 发行版，以常见的 Ubuntu 来说，可以使用 16.04 版，如果是 CentOS 则可以使用 CentOS 7，使用较高版本的系统，可以省去很多额外安装依赖的麻烦，让我们一步步开始。

（1）安装 Docker 运行环境

Docker 是一个轻量级的容器环境，类似于虚拟机，但是比虚拟机要轻很多，在一个操作系统中可以运行相当多数量的 Docker 容器，每一个容器中可以运行独立的服务，容器与容器之间是隔离的，不会互相有干扰，通常在安装 Docker 的时候都会连带一起安装 Docker Compose，通过 Docker Compose 可以方便地部署多个 Docker 容器实例，我们将使用这些工具来部署 Fabric 节点，每一个节点都运行在一个独立的 Docker 容器中。安装时请选择不低于 1.12 的版本，安装完毕后，可以分别运行如下命令检查 Docker 和 Docker Compose 的版本：

```
docker -v
docker-compose -v
```

（2）安装 Go 运行环境

Fabric 使用 Go 语言开发，因此需要安装 Go 的环境，这个也不再赘述了，安装完毕后不要忘记设置 GOPATH 环境变量。分别使用如下命令查看 Go 的版本以及 GOPATH 的设置：

```
go version
echo $GOPATH
```

（3）安装 Node.js 以及 NPM

Fabric 提供有多种语言版本的 SDK，可以用于通过 API 的调用与 Fabric 构建的区块链服务进行交互，我们使用 Fabric 提供的针对 Node.js 的 SDK 来开发应用，注意保持 Node 运行时的版本为 6.9.x，这里使用的版本是 6.9.5，目前官方的 SDK 还没有支持更高版本的 Node，通过以下命令可以查看 Node 安装的版本：

```
node -v
npm -v
```

至此，基础环境就安装完毕了，接下来安装 Fabric。

（4）创建目录

下载官网提供的平台相关的二进制文件到创建的目录中。

```
cd ~ | mkdir fabricsample
cd ~/fabricsample
```

（5）下载 Fabric 组件

```
curl -sSL https://goo.gl/iX9dek | bash
```

使用 curl 工具下载 Fabric 的组件，使用这些文件就可以设置 Fabric 网络了，下载完毕后，可以看到在目录中生成了一个 bin 目录，进去后可以看到有以下的文件：



我们解释一下几个组件的作用：

- ❑ configtxgen：用于生成共识服务启动以及通道创建所需的配置数据，它需要一个名为 configtx.yaml 的配置文件，在这个文件中包含了 Fabric 节点网络的定义。
- ❑ configtxlator：可以用来将通道配置信息转换为可读形式。
- ❑ cryptogen：用于生成 x509 标准证书，用于实现身份识别等鉴权功能，这个命令工具需要使用一个名为 crypto-config.yaml 的配置文件，在这个配置文件中包含需要部署的 Fabric 网络拓扑结构，根据网络结构，cryptogen 命令可以生成所需的证书库以及密钥。
- ❑ get-byfn.sh：用来下载名为 byfn 的脚本程序，实际上其中包含了一个下载命令。
- ❑ get-docker-images.sh：这是一个脚本程序，可以用来下载 Fabric 的各个组件到本地的 Docker 容器中。
- ❑ orderer：共识服务程序，在超级账本中将共识服务独立为一个节点程序，负责将网络中的交易事务打包进区块，并使用通道机制订阅给其他的对等节点（也可以说是账本节点）。
- ❑ peer：账本节点程序，用于维护账本数据并且运行智能合约，以下统称这种节点为对等节点。

我们可以看到，运行超级账本的 Fabric，要比运行比特币或者以太坊复杂很多，作为一个商业级的联盟链基础设施，增加了很多公链系统没有的组件功能。

除了下载这些文件，命令脚本同时还安装了 Fabric 组件的 Docker 镜像，从 Docker

Hub 下载到本机的 Docker 仓库，使用 docker images 可以查看到，分别是：

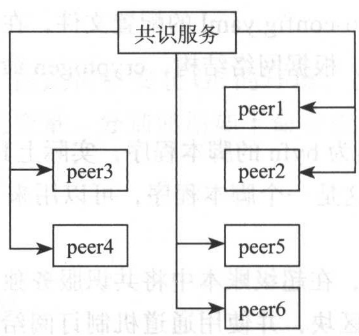
镜像	作用	镜像	作用
hyperledger/fabric-orderer	共识服务节点	hyperledger/fabric-ccenv	智能合约环境
hyperledger/fabric-peer	对等节点	hyperledger/fabric-javaenv	支持 Java 的智能合约环境
hyperledger/fabric-couchdb	状态存储库	hyperledger/fabric-tools	Fabric 工具库
hyperledger/fabric-kafka	分布式消息队列	hyperledger/fabric-ca	Fabric 证书管理组件
hyperledger/fabric-zookeeper	分布式协调服务		

这里需要对 Fabric 的结构和模块做一些介绍。在 Fabric 中，对区块链应用中的各个角色进行了明确的划分，这也是其高度模块化的一个体现，不再像比特币这样一股脑儿都混在一起，Fabric 面向商业应用，因此其本质上是设计为一个私有链或者说是联盟链，除了通常的模块元素外，主要特点表现在以下几个方面。

1）具有多种类型的节点，比如负责管理账本数据的 peer 对等节点，负责提供共识服务的 orderer 共识服务节点，负责鉴权的身份服务节点，负责创建和校验交易并且维护智能合约状态的验证节点，负责提供用户端服务的应用节点等。

2）对等节点之间的账本数据共享通过一个称为 channel（也就是通道）的机制来过滤，通道是 Fabric 中一个富有特色的机制，正是通过通道的概念，实现了数据的隔离分发，只有处于同一个通道的节点之间才会分享账本数据，通过这种机制，在同一个联盟链的对等节点之间，可以根据策略拥有不同的账本副本数据。

注意，对于共识服务节点来说，是接收所有数据的，通道只是与对等节点相关，实际上对等节点是通过向共识服务节点订阅了不同的主题，而每个主题就是一个通道，它们的关系如下图所示：



图中 peer1 与 peer2 订阅了同一个通道，peer3 与 peer4 订阅了同一个通道，peer5 与 peer6 订阅了同一个通道，根据不同的通道订阅，共识服务根据策略分发不同的区块数据，我们可以发现，在 Fabric 的设计中充分考虑了作为商业环境使用的安全问题。

通过证书颁发服务进行身份认证与鉴权，这个与传统的企业级系统是类似的，限制进入系统的用户，设置不同的权限，作为面向商业使用的系统，这个显然是必备的，也是与公链系统（如比特币、以太坊等）很大的一个差别。

Fabric 由于有通道和身份认证的设施，使得对于每一个节点看到的数据都是可以不一样的，也就是说从逻辑上来看，Fabric 是一个实现了多通道多链结构的一个区块链网络。这是很有意思的，此前的比特币、以太坊等公链系统，每个节点看到的数据都是一样的，无论是实际的物理数据还是逻辑上的视图数据都是一致的。Fabric 的这个特点，类似于数据库系统中的物理表与视图的概念，通过设定不同的视图逻辑，实现不同的数据管控要求。

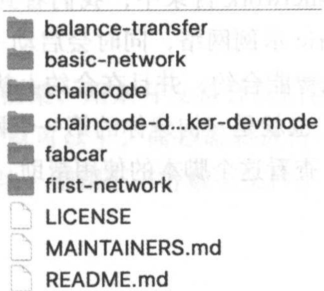
7.3　Fabric 示例

7.3.1　部署准备

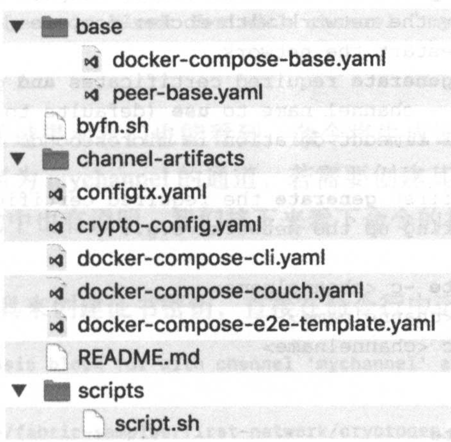
1. 下载示例程序

我们将当前的工作目录切换到 fabricsample 目录中，下载官网提供的示例：git clone https://github.com/hyperledger/fabric-samples.git。

下载完毕后，在目录下多了一个 fabric-samples 目录，可看到如下文件：

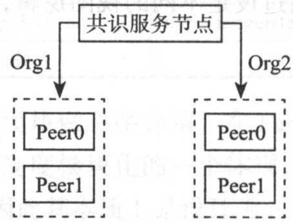


其中包含了好几个示例，我们选择其中的 first-network 来做测试，进入到 first-network 目录，看到如下一组文件：



这里面大多数是 yaml 配置文件以及两个脚本文件：byfn.sh 和 script.sh。Fabric 组件的运行需要使用到这些配置文件，而两个 sh 脚本则用来控制 Fabric 组件的运行。

通过查看配置文件，可以发现这是一个多节点 Fabric 网络示例，包含了 4 个对等节点以及 1 个共识服务节点，4 个对等节点分成了 2 个组织域，大致是如下的运行示意图：



注意，图中的示例是指分成两个组，而不是两个通道。在这个示例中，4 个节点共用一个通道。

2. 查看帮助

到现在为止，我们已经安装了示例程序运行所需的基础环境，接下来就可以试一试这个 first-network 了，进入到 first-network 目录中，我们看到有一个脚本程序 byfn.sh，通过运行这个脚本可以启动这个 Fabric 示例网络，同时会启动一个容器用来执行脚本在通道中加入新的节点以及部署和初始化智能合约，并且在合约上执行交易。文件名 byfn 其实就是 build your first network 的缩写，也就是“构建你的第一个网络”的意思。大家也可以在终端命令行中通过命令 ./byfn.sh -h 查看这个脚本的使用帮助，执行结果如下：

```
./byfn.sh -h

Usage:
  byfn.sh -m up|down|restart|generate [-c <channel name>] [-t <timeout>]
  byfn.sh -h|--help (print this message)
    -m <mode> - one of 'up', 'down', 'restart' or 'generate'
      - 'up' - bring up the network with docker-compose up
      - 'down' - clear the network with docker-compose down
      - 'restart' - restart the network
      - 'generate' - generate required certificates and genesis block
    -c <channel name> - channel name to use (defaults to "mychannel")
    -t <timeout> - CLI timeout duration in microseconds (defaults to 10000)

Typically, one would first generate the required certificates and
genesis block, then bring up the network. e.g.:

    byfn.sh -m generate -c <channelname>
    byfn.sh -m up -c <channelname>
    byfn.sh -m down -c <channelname>

Taking all defaults:
    byfn.sh -m generate
```

```
byfn.sh -m up
byfn.sh -m down
```

命令中包含了启动网络、清除网络、重启网络以及生成证书和创世区块等使用说明，事实上详细的各个命令参数具体是怎么运行的，也可以直接打开 byfn.sh 源码来查看，这就是一个 bash 脚本程序，实际上这个脚本就是通过调用我们下载的 Fabric 组件程序以及示例代码的配置文件来部署整个示例网络的。

```
// 生成证书与创世区块
byfn.sh -m generate

// 启动部署在 docker 容器中的 fabric 网络
byfn.sh -m up

// 停止并清除运行在 docker 容器中的 fabric 组件
// docker 中的镜像并不删除，相当于 byfn.sh -m up 的逆过程
byfn.sh -m down

// 重启 fabric 网络
byfn.sh -m restart
```

3. 数据配置

Fabric 是一套半成品的开发框架，用来开发符合我们需求的区块链系统，因此不像比特币、以太坊直接下载下来运行就可以了，而是需要进行一系列的数据配置，按照 Fabric 的运行要求，需要设定好创世区块、密钥证书等数据文件。

（1）生成证书和创世区块

```
./byfn.sh -m generate
```

执行这个命令需要用到 configtxgen 和 cryptogen，因此别忘了把这两个命令程序复制到 first-network 目录中，执行过程中会有提示：

```
Generating certs and genesis block for with channel ‘mychannel’ and CLI timeout
of ‘10000’
Continue (y/n)? y
```

按下 y 键继续即可，通过提示我们也能看到，命令将生成证书和创世区块，同时也可以看到，命令默认创建了名为 mychannel 的通道，若需要创建其他的名称，可以使用 -c 参数指定，在上述的命令帮助中也有说明。我们接下来看下命令的执行过程输出。

（2）生成证书及密钥

我们使用 cryptogen 工具来创建证书密钥，直接在命令行中运行即可，如下所示：

```
Generating certs and genesis block for with channel 'mychannel' and CLI timeout of '10000'
Continue (y/n)? y
proceeding ...
/Users/apple/fabricsample/fabric-samples/first-network/cryptogen
```

```
############################################################
##### Generate certificates using cryptogen tool #########
############################################################
org1.example.com
org2.example.com
```

命令执行后，可以看到结果提示，其中 org1.example.com 与 org2.example.com 是创建的两个对等节点组织的域名，在 Fabric 网络中，节点是由节点组织来管理的，无论是普通的对等节点还是共识服务节点，节点组织具有组织名和域名，本例中定义了如下的组织关系：

组织名	组织域名	节点类型	节点主机名	节点全名
Org1	org1.example.com	对等	peer0	peer0.org1.example.com
Org1	org1.example.com	对等	peer1	peer1.org1.example.com
Org2	org2.example.com	对等	peer0	peer0.org2.example.com
Org2	org2.example.com	对等	peer1	peer2.org2.example.com
orderer	example.com	共识	orderer	orderer.example.com

这些生成配置都定义在 crypto-config.yaml 中，我们看下 crypto-config.yaml 的主要内容：

```
//共识节点组织定义
OrdererOrgs:
  - Name: Orderer  //共识节点名称
    Domain: example.com //域名
    Specs:
      - Hostname: orderer
//对等节点组织定义
PeerOrgs:
  - Name: Org1  //第一个对等节点组织
    Domain: org1.example.com  //域名
    Template:
      Count: 2
    Users:
      Count: 1

  - Name: Org2  //第二个对等节点组织
    Domain: org2.example.com //域名
    Template:
      Count: 2
    Users:
      Count: 1
```

通过文件内容，可以看到有节点组织以及包含的节点数、节点名称和域名等配置信息。补充一点对于节点名称和域名的关系，在 Fabric 中，一个节点的名称组成如下：

```
{Hostname}.{Domain}
```

比如 org1 组织下管理两个节点，分别是 peer0 与 peer1，则这两个 peer 节点的全名是：

```
peer0.org1.example.com
peer1.org1.example.com
```

在生成证书时，会为节点组织以及组织下的每个节点都生成一系列的证书，生成的数字证书和私钥都存储在 crypto-config 目录中，打开可以看到目录结构：



每个组织都会生成一个唯一的根证书 ca-cert，使用根证书绑定节点（对等节点与共识节点），加入链的成员可以使用自己的证书进行获取授权。交易与通信使用节点的私钥签名，验证则使用公钥，这便是 Fabric 中的证书体系了，实际上就是一套公钥设施。

（3）生成创世区块

准备好证书密钥这些配置数据后，我们就可以开始了，首先创建创世区块，这是使用 configtxgen 工具来实现的，在命令行中执行此命令，如下所示：

```
/Users/apple/fabricsample/fabric-samples/first-network/configtxgen
##########################################################
#########  Generating Orderer Genesis block ##############
##########################################################
2017-07-20 08:52:03.116 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-07-20 08:52:03.140 CST [common/configtx/tool] doOutputBlock -> INFO 002 Generating genesis block
2017-07-20 08:52:03.141 CST [common/configtx/tool] doOutputBlock -> INFO 003 Writing genesis block
```

生成的创世区块用来启动共识服务节点，共识服务节点首先拥有区块数据，然后使用通道与 peer 节点之间进行数据同步。以下的“通道配置事务”与“锚节点”也都是通过 configtxgen 命令程序生成的。

（4）生成通道配置事务

```
#################################################################
### Generating channel configuration transaction 'channel.tx' ###
#################################################################
2017-07-20 08:52:03.157 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-07-20 08:52:03.160 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO 002 Generating new channel configtx
2017-07-20 08:52:03.160 CST [common/configtx/tool] doOutputChannelCreateTx -> INFO 003 Writing new channel tx
```

主要是对通道的事务规则配置。

（5）生成锚节点

```
#################################################################
#######    Generating anchor peer update for Org1MSP   ##########
#################################################################
2017-07-20 08:52:03.174 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-07-20 08:52:03.177 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 002 Generating anchor peer update
2017-07-20 08:52:03.177 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing anchor peer update

#################################################################
#######    Generating anchor peer update for Org2MSP   ##########
#################################################################
2017-07-20 08:52:03.194 CST [common/configtx/tool] main -> INFO 001 Loading configuration
2017-07-20 08:52:03.197 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 002 Generating anchor peer update
2017-07-20 08:52:03.197 CST [common/configtx/tool] doOutputAnchorPeersUpdate -> INFO 003 Writing anchor peer update
```

锚节点是指通道中能被所有其他对等节点发现，并能进行通信的一种对等节点。通道中的每个成员都有一个（或多个，以防单点故障）锚节点，允许属于不同成员身份的节点来发现通道中存在的其他节点，相当于节点中的网关。

7.3.2 启动 Fabric 网络

经过了上述的准备工作后，就可以开始启动 Fabric 网络了，使用如下命令：

```
./byfn.sh -m up
```

这个命令做的事情不少，需要那么一段执行时间，而且一不小心还会有如下错误提示：

```
ERROR: manifest for hyperledger/fabric-tools:latest not found
```

这是因为 byfn.sh 命令要求 Docker 容器安装的 Fabric 组件的 tag 是 latest 标签的，不过也无妨，如果组件安装是正常的，也可以直接修改 tag。如果有上面这样的错误提示，可以使用 Docker tag 命令处理，命令格式如下：

```
docker tag 0403fd1c72c7 hyperledger/fabric-tools:latest
```

命令中的 0403fd1c72c7 是 hyperledger/fabric-tools 在 Docker 中的镜像 ID，读者遇到类似问题时，可以采用上述命令的方法修正，注意要替换成自己 Docker 容器中对应组件的镜像 ID，其他的 Fabirc 组件也是同样的处理方式，修正了 tag 之后，重新开始执行 ./byfn.sh -m up，可以看到执行了如下的步骤：

序号	过程	作用
1	Creating peer0.org2.example.com	创建 org2 中的 peer0 对等节点
2	Creating peer1.org1.example.com	创建 org1 中的 peer1 对等节点

（续）

序号	过程	作用
3	Creating orderer.example.com	创建名为 orderer 的共识节点
4	Creating peer1.org2.example.com	创建 org2 中的 peer1 对等节点
5	Creating peer0.org1.example.com	创建 org1 中的 peer0 对等节点
6	Channel "mychannel" is created	创建名为 mychannel 的通道
7	Having all peers join the channel	将对等节点加入到通道
8	Updating anchor peers for org1	更新 org1 中的锚节点为 peer0.org1.example.com
9	Updating anchor peers for org2	更新 org2 中的锚节点为 peer0.org2.example.com
10	Install chaincode on org1/peer0	在 org1/peer0 节点上安装智能合约
11	Install chaincode on org2/peer2	在 org2/peer2 节点上安装智能合约
12	Instantiating chaincode on org2/peer2	实例化 org2/peer2 上的智能合约
13	Querying chaincode on org1/peer0	访问 org1/peer0 上的智能合约
14	Sending invoke transaction on org1/peer0	在 org1/peer0 上发起调用交易事务
15	Install chaincode on org2/peer3	在 org2/peer3 节点上安装智能合约
16	Querying chaincode on org2/peer3	访问 org2/peer3 上的智能合约

执行完毕后，输出一个结束符号：

```
========= All GOOD, BYFN execution completed ===========

 _____   _   _   ____
| ____| | \ | | |  _ \
|  _|   |  \| | | | | |
| |___  | |\  | | |_| |
|_____| |_| \_| |____/
```

至此，Fabric 示例网络就运行起来了，上述过程中的 org2/peer2 就是指 peer0.org2.example.com，org2/peer3 就是指 peer0.org2.example.com，由于两个组织域中分别有 peer0 与 peer1 两个对等节点，为了便于称呼，将这 4 个对等节点依次称为 org1/peer0、org1/peer1、org2/peer2、org2/peer3。

容器 ID	镜像名称	端口	容器名称
f13b6eb5987c	dev-peer1.org2.example.com-mycc-1.0		dev-peer1.org2.example.com-mycc-1.0
bb99b6657ca2	dev-peer0.org1.example.com-mycc-1.0		dev-peer0.org1.example.com-mycc-1.0
beace0743b25	dev-peer0.org2.example.com-mycc-1.0		dev-peer0.org2.example.com-mycc-1.0
edb83ef24b9c	hyperledger/fabric-peer	0.0.0.0:10051->7051/tcp, 0.0.0.0:10053->7053/tcp	peer1.org2.example.com
b2be7d40a31f	hyperledger/fabric-peer	0.0.0.0:7051->7051/tcp, 0.0.0.0:7053->7053/tcp	peer0.org1.example.com

（续）

容器 ID	镜像名称	端口	容器名称
80ab913bb7d9	hyperledger/fabric-peer	0.0.0.0:8051->7051/tcp, 0.0.0.0:8053->7053/tcp	peer1.org1.example.com
3e5e4189ca8a	hyperledger/fabric-peer	0.0.0.0:9051->7051/tcp, 0.0.0.0:9053->7053/tcp	peer0.org2.example.com
6268882b5bfe	hyperledger/fabric-orderer	0.0.0.0:7050->7050/tcp	orderer.example.com
41514af0bd1d	hyperledger/fabric-tools		cli

可以看到，安装到Docker中的Fabric节点已经运行起来了，其中peer是指对等节点，orderer是指共识节点，除了5个节点容器外，还有前面3行是指智能合约容器，系统会为节点上的智能合约操作启动一个容器，最后一个是Fabric工具组件，也启动在一个容器中。

我们通过docker logs来查看下一个智能合约的容器日志。

```
docker logs dev-peer0.org1.example.com-mycc-1.0
// 输出
ex02 Invoke
Query Response:{"Name":"a","Amount":"100"}
ex02 Invoke
Aval = 90, Bval = 210

docker logs dev-peer0.org2.example.com-mycc-1.0
// 输出
ex02 Init
Aval = 100, Bval = 200

docker logs dev-peer1.org2.example.com-mycc-1.0
// 输出
ex02 Invoke
Query Response:{"Name":"a","Amount":"90"}
```

通过输出的容器日志，我们能看到在启动网络后运行智能合约的操作所带来的Aval与Bval两个资产金额的变化，要更详细地了解运行的智能合约的内容，请参见7.3.3节。

7.3.3 Fabric智能合约

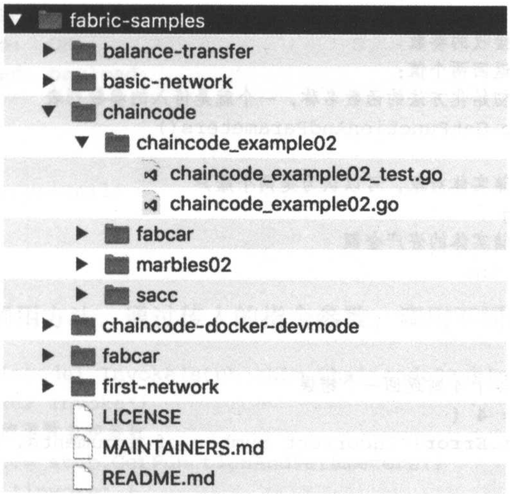
通过前面的操作，整个启动程序完成了一系列的动作，并部署了智能合约，那么这份智能合约是什么样的呢？Fabric用来搭建商用联盟链系统，而所谓的商用，首要的就是智能合约的应用，本节我们就来分析一下部署在合约中的源码文件，一窥Fabric中智能合约的究竟。这份源码文件就在我们下载的示例程序目录中的chaincode文件夹中。

其中chaincode_example02.go就是合约的源码文件，合约代码是使用Go语言编写的，我们分段来看下这份合约代码。

（1）引入包

这是Go语言中的机制，类似于C语言中的引入头文件或者Java中的导入包，目的是

将已经具备的一些功能代码直接导入进来，也可以简单地理解为现成的功能库，如下所示：




```
package main
import (
    "fmt"
    "strconv"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
)
```

除了标准的 fmt 与 strconv 外，后两个包都是在 Fabric 源码中定义的，其中的 shim 包是用于访问智能合约的接口定义，最后一行引入的是与对等节点通信相关的 Prototuf 定义，Protobuf 是 Google 提供的一个开源序列化框架，类似于 XML、JSON。

（2）定义结构类型

该段代码定义了一个名为 SimpleChaincode 的结构类型，后续都是定义在这个结构类型上的方法，这是 Go 语言中类似于 Java 和 C++ 中类的概念。

```
// SimpleChaincode example simple Chaincode implementation
type SimpleChaincode struct {
}
```

（3）合约初始化方法

智能合约是用来定义一套规则的，而规则归根结底是用来在某个条件下更新合约中定义的数据的，比如资产金额等，因此我们在使用智能合约之前，就得先把这些数据给初始化了，初始化后的合约会写入在区块链账本中。

```
// 参数 ChaincodeStubInterface 是一个接口定义，用于部署合约的应用访问和修改账本数据
// 这个接口定义在 fabric 源码的 fabric/core/chaincode/shim/interfaces.go 文件中
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response
```

```
{
    fmt.Println("ex02 Init")

    // 获得初始化方法接收的参数
    // 这个方法实际是返回两个值:
    // 一个是调用这个初始化方法的函数名称，一个就是传入的参数列表
    _, args := stub.GetFunctionAndParameters()

    // 定义两个变量存储实体对象，可以认为是两个账户
    var A, B string
    // 定义两个变量存储实体的资产金额
    var Aval, Bval int

    var err error

    // 如果参数个数不等于 4 则返回一个错误
    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting 4")
    }

    // 第 1 个参数赋值为实体对象 A
    A = args[0]
    // 第 2 个参数为实体对象 A 的资产金额
    Aval, err = strconv.Atoi(args[1])
    if err != nil {
        return shim.Error("Expecting integer value for asset holding")
    }

    // 第 3 个参数赋值为实体对象 B
    B = args[2]
    // 第 4 个参数为实体对象 B 的资产金额
    Bval, err = strconv.Atoi(args[3])
    if err != nil {
        return shim.Error("Expecting integer value for asset holding")
    }

    // 控制台输出两个资产金额
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

    // 将实体 A 及初始资产金额更新到账本数据
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    // 将实体 B 及初始资产金额更新到账本数据
    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    // 返回一个 json 格式的成功响应
```

```
    /*
    func Success(payload []byte) pb.Response {
        return pb.Response{
            Status: OK,
            Payload: payload,
        }
    }
    */
    return shim.Success(nil)
}
```

（4）合约调用

以下是合约操作的调用方法，通过传入的指令参数，调用不同的操作方法。

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Invoke")
    //获得调用本方法的函数名和参数
    function, args := stub.GetFunctionAndParameters()
    if function == "invoke" {
        //若调用方法名为 invoke，则调用一个执行从 A 到 B 的转账交易的方法
        return t.invoke(stub, args)
    } else if function == "delete" {
        //若调用方法名为 delete，则表示调用一个删除实体对象的方法
        return t.delete(stub, args)
    } else if function == "query" {
        //若调用方法名为 query，则表示调用一个查询实体对象的方法
        return t.query(stub, args)
    }
    //若没有找到对应的方法，则返回一个"无效调用方法名称"的错误提示
     return shim.Error("Invalid invoke function name. Expecting \"invoke\"
\"delete\" \"query\"")
}
```

（5）转账交易调用

这份合约中定义的就是 A 和 B 两个账户之间的资产金额转账，因此必须给出实现这个功能的方法，比如从 A 转账到 B，其过程就是先获得 A 的金额，然后写入到 B，再从 A 中扣除等额的转账金额，最后将这些变更更新到区块链账本中。我们看到，除了基于区块链这一点外，业务逻辑的实现与普通程序没有什么不同，我们看一下代码实现：

```
// 本调用方法实现从 A 转账一定数额到 B 的交易
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
    var A, B string     // 定义 A 与 B 两个实体对象
    var Aval, Bval int // 定义两个变量分别存储 A 与 B 的资产金额
    var X int          // 交易金额
    var err error

    //若传入的参数个数不等于 3，则报出错误提示
    if len(args) != 3 {
```

```
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }

    // 分别将第一个参数与第二个参数赋值给 A 与 B，这是传入的两个实体对象
    A = args[0]
    B = args[1]

    // 从账本中获得 A 当前的资产金额，若之前没有发生过交易则就是初始化的金额
    Avalbytes, err := stub.GetState(A)
    if err != nil {
        return shim.Error("Failed to get state")
    }
    if Avalbytes == nil {
        return shim.Error("Entity not found")
    }
    Aval, _ = strconv.Atoi(string(Avalbytes))

    // 从账本中获得 B 当前的资产金额，若之前没有发生过交易则就是初始化的金额
    Bvalbytes, err := stub.GetState(B)
    if err != nil {
        return shim.Error("Failed to get state")
    }
    if Bvalbytes == nil {
        return shim.Error("Entity not found")
    }
    Bval, _ = strconv.Atoi(string(Bvalbytes))

    // 传入的第三个参数为交易金额
    X, err = strconv.Atoi(args[2])
    if err != nil {
            return shim.Error("Invalid transaction amount, expecting a integer
value")
    }
    // 从 A 转账给 B，因此 A 的金额减掉 X，B 的金额加上 X
    Aval = Aval - X
    Bval = Bval + X
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

    // 将 A 交易后金额的变更写入到账本中
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    // 将 B 交易后金额的变更写入到账本中
    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    // 返回成功
```

```
        return shim.Success(nil)
    }
```

（6）从账本中删除实体对象

这段代码实现的功能，是在不需要保留合约中某个对象时进行删除，比如不需要 A 账户或者 B 账户时可以删除掉，当然实际商业环境中的智能合约，要删除合约中定义的某个对象肯定需要一些条件或者验证，这里只是一个功能示例。

```
    func (t *SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
        //若没有参数则报错
        if len(args) != 1 {

            return shim.Error("Incorrect number of arguments. Expecting 1")
        }

        //从参数中获得需要删除的实体对象
        A := args[0]

        //执行删除
        err := stub.DelState(A)
        if err != nil {
            return shim.Error("Failed to delete state")
        }

        return shim.Success(nil)
    }
```

（7）余额查询

功能很简单，就是查询合约中定义的账户对象的余额，由于查询并不会改动合约中的数据对象，因此直接返回结果即可。

```
    func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
        var A string //定义一个变量存储实体对象
        var err error

        //若没有参数则报错
        if len(args) != 1 {
            return shim.Error("Incorrect number of arguments. Expecting name of the
person to query")
        }
        //将参数中的实体对象赋值给A
        A = args[0]
        //从账本中获得A的金额
        Avalbytes, err := stub.GetState(A)
        //若错误对象不为空则报错
        if err != nil {
            jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
```

```
            return shim.Error(jsonResp)
        }
        //若金额为空则返回金额为空的结果
        if Avalbytes == nil {
            jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
            return shim.Error(jsonResp)
        }
        //以json格式返回余额结果
         jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) +
"\"}"
        fmt.Printf("Query Response:%s\n", jsonResp)

        return shim.Success(Avalbytes)
    }

    //合约的入口启动方法，实例化后启动
    func main() {
        err := shim.Start(new(SimpleChaincode))
        if err != nil {
            fmt.Printf("Error starting Simple chaincode: %s", err)
        }
    }
```

至此，我们对示例合约代码就作了一个简单的注释分析，可以看到，这是一份功能非常简单的智能合约，创建资产对象 / 初始化 / 转账交易 / 删除对象 / 查询余额，就是这些基本的功能，这就是这个 Fabric 示例网络所部署的智能合约。

了解了智能合约的内容后，我们再来看一个命令行操作，根据上述的步骤我们已经启动了示例网络，那么再来访问下节点的智能合约，我们知道了这份智能合约的功能就是两个实体对象之间的资产金额管理，执行一个查询操作，首先进入 fabric-tools 的容器命令环境：

```
docker exec -it "cli"  /bin/bash
```

进入命令环境后，查询一下 a 与 b 的资产金额：

```
peer chaincode query -C "mychannel" -n mycc -c '{"Args":["query","a"]}'
peer chaincode query -C "mychannel" -n mycc -c '{"Args":["query","b"]}'
```

通过输出的结果可以看到 a 的金额是 90，b 的金额是 210，再做一次转账操作，从 a 转 10 到 b：

```
homeaddr=/opt/gopath/src/github.com/hyperledger/fabric/peer
ordererercertaddr=/crypto/ordererOrganizations/example.com/orderers/orderer.
example.com/msp/tlscacerts/tlsca.example.com-cert.pem

peer chaincode invoke -o orderer.example.com:7050
--tls    $CORE_PEER_TLS_ENABLED
--cafile $homeaddr$orderercertaddr
-C "mychannel" -n mycc
```

```
-c '{"Args":["invoke","a","b","10"]}'
```

执行后，输出一个提示 " Chaincode invoke successful. result: status:200"，这就表示转账交易调用成功了，按照计算，现在 a 应该是有 80，b 有 220，我们再来查询一下余额，查询方法与上同，可以看到输出的结果分别是：

```
Query Result: 80
Query Result: 220
```

到现在为止，我们已经完整地体验了一回 Fabric 网络的部署和使用以及智能合约代码的逻辑，在示例中我们只是使用了官方提供的测试用合约代码，读者感兴趣可以自行修改示例中的代码，体会一下基于 Fabric 的智能合约开发。

7.3.4 Fabric 部署总结

基于上述步骤，可以知道通过 Fabric 部署一个智能合约的节点网络，大体上需要经过如下的步骤：

1）生成必要的文件，比如节点证书、创世区块、通道事务配置、锚节点
2）创建通道
3）加入节点到通道中
4）更新锚节点
5）安装智能合约
6）实例化智能合约
7）调用执行智能合约

从部署步骤来看，与以太坊是类似的，只不过多了一些证书、通道、锚节点等额外的修饰功能，通过使用 Fabric 组件，我们部署一个智能合约的过程相当简单，基本上主要工作只是编写智能合约文件，其他的基础设施功能都提供好了，通过 SDK 可以进行节点功能的调用。这些对于普通的区块链应用开发者来说，就像是入住酒店，除了带自己的必需物品外，其他设施一应俱全了。

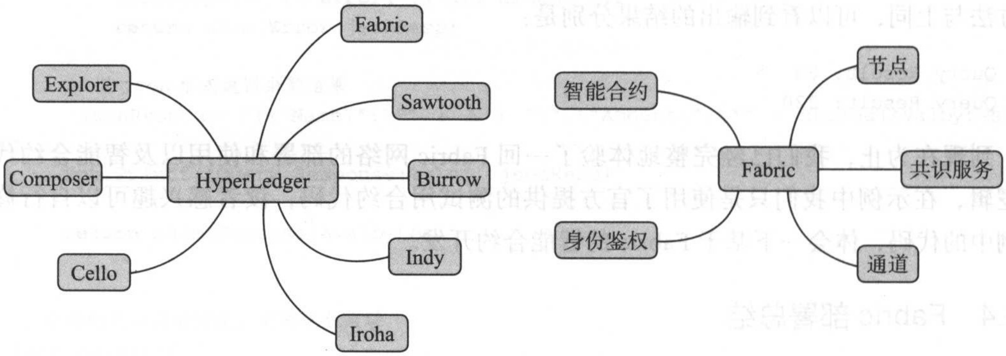
另外，超级账本项目中并不只是一个 Fabric，其他的各个子项目也值得去学习试用一番，这些都是国际大公司贡献的代码。对于技术开发人员来说，仔细阅读理解其中的源码以及文档是大有裨益的，对于区块链底层设施的设计能有很好的提升。

7.4 知识点导图

超级账本项目是面向商业应用的，其中的 Fabric 项目可以认为是对标以太坊的区块链系统，面向企业应用，考虑了很多更复杂的特性，比如身份认证、通道等，目的是提高数据网络的安全性。但是有一点需要注意，从技术角度来说，Fabric 只是一个技术框架，并不

是一个像比特币、以太坊这样的公链系统，我们可以通过使用 Fabric 来搭建自己需要的区块链应用系统，自己来部署节点，这是一个很大的区别。

我们来看下知识点导图。



第 8 章 Chapter 8

动手做个实验：搭建微链

区块链程序作为一种计算机软件，如果去除掉那些外部的修饰，抹掉那些思想层面的、金融层面的、哲学层面的包装，它就只是一个普通的应用程序，与我们日常使用的聊天软件、游戏软件、视频播放软件等一样，没什么特别的。如今大家也可以看到有各种各样基于区块链设计的软件，就如前面章节所述，有些是独立的应用系统，如比特币、以太坊等；有些是基于已有系统开发的，如各种以太坊代币应用；有些是提供了区块链资产交易功能的，如比特股、公信宝等；有些是面向开发者服务的，如万云区块链云平台、布比区块链等。技术发展如此之快，我们上下求索唯恐不及，能不能先不要让人眼花缭乱，给一个简单的例子？是的，这就是本章的目的，让我们站在代码的角度，看看一个最简单的区块链程序是怎样组成的。

8.1 微链是什么

毫无疑问，要开发一个完整可用的区块链应用程序，不是那么容易的，大家看看比特币的源码、以太坊的源码等就知道了。比特币作为第一代区块链技术的代表，其功能设计比较简单，即便如此，相信不少初次阅读源码的读者仍然会觉得有些迷茫。微链的目的就是以比特币为原型，假设我们自己要开发一个比特币程序（或称为微币），会怎么去做，通过一个极简的结构设计说明，以俯瞰的方式来了解一个区块链应用程序的基本构造。通过微链的设计，我们至少可以回答以下问题：

- 一个区块链应用程序需要包含哪些基本模块？
- 一代技术（如比特币）与二代技术（如以太坊）主要有哪些区别？

- ❑ 钱包、挖矿、区块链账本等到底是怎样的组合关系？
- ❑ 所谓的可编程数字货币到底是什么意思？
- ❑ 区块链应用程序可以作为单机程序运行吗？

*微链会采用半源码半伪码的方式来进行说明，功能模块参照比特币。有读者说，为什么不直接模拟设计一个二代技术产品（如以太坊）来讲解呢？*以太坊支持的功能更加强大，不但支持数字货币还支持各种智能合约的编写，目前应用也很广泛，这个算是现在的主流技术了。我们从一个简单的开始，了解清楚主要的程序结构组成，再去理解更复杂的以太坊等其他各种区块链应用，也就心中有数了。

时常有人问起，到底什么是区块链？网络上有很多资料，各种看不明白的名词，一会儿说去中心化，一会儿又说其实不能叫去中心化而应该叫分布式，一会儿又是各种应用代币、去中心交易所、区块链操作系统，看得眼花缭乱，让人感觉很神秘，好像任何东西只要兑上点区块链这款药水，立马就能腐朽变神奇，立马就变成了能解救人类难题的法宝。其实我们知道，区块链技术并不是什么基因突变出来的未来技术，事实上组成区块链技术的各个部分，在计算机发展领域中早就有了，比如哈希计算、公开密钥加密技术、点对点网络通信，这些都是早已在运用的常规软件开发技术了。区块链应用真正值得称道的是将这些传统而成熟的技术巧妙组合在一起，实现了一个非常有意思的功能。看过本章微链的介绍后，大家也就知道区块链技术真正的伟大之处了，不是在技术上，而是在思想上。

8.2 开发环境准备

区块链程序本质上与普通的软件是一样的，因此在开发方式上并没有什么特别的区别。

从开发语言上来说，但凡是图灵完备的语言都可以用来开发，比如 C++/Java/Go/ 等，还有现在比较流行的 Node. js，目前来说，生产环境的正式程序开发，使用 C++ 和 Go 比较多，比如比特币就是使用 C++ 开发的，以太坊是使用 Go 开发的（以太坊同时也有其他语言的版本，如 C++、Python），而一些测试环境的程序会使用 Node. js，比如模拟以太坊环境的 testrpc 程序，testrpc 是使用 JavaScript 开发的，并且以 Node 包的形式发布。可以说，语言选择上没什么限制，选择自己熟悉的即可，个人比较推荐 Go，运行效率不错，且 Go 本身是运行在虚拟机上的，因此也跨平台，语法也容易上手，不过 Go 目前不太适合开发图形界面，好在区块链核心程序本来也不需要界面，各种带界面的客户端可以使用 WebApp 的方式来搭建。

从操作系统环境上来说，比较推荐 Linux 或者 Mac 系统，这个就不再赘述了。

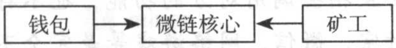
开发工具没什么特别要求，以微链来说，是使用 go 语言来说明的。编辑器使用 Visual Studio Code 或者 Vim 皆可，编译则可以直接使用 Go 的编译命令 go build，如果嫌麻烦，也可以使用一款叫 LiteIDE 的开源集成开发环境。关于 Go 语言的安装和配置这里就不赘述了，官网有很详细的说明。

开发环境的说明基本就是这样了，大家在实验编写代码时，不用一开始就考虑太多的关于优化或者代码结构组织等方面的问题，我们的目的是理解区块链代码而不是优化区块链代码，这些事等真正开发应用的时候再考虑不迟。另外，带着实验、玩乐的心态来摆弄区块链是最有意思的，很多时候，阻碍我们去创新的，不是技术，而是应用思想。

8.3 设计一个简单的结构

现在，我们先来看一下微链打算展示哪些功能设计，如下：

- 具备一个微链核心，支持同步区块数据并验证和存储区块数据到主链。
- 具备一个钱包功能，可以存储公钥私钥以及账户余额。
- 具备一个挖矿功能，用于打包区块数据并发行新的微币。



到这里就结束了，对于一个区块链应用程序来说，大的功能模块其实就是这些，其他各种功能（如区块数据同步、数据完整性验证、解锁与锁定脚本等），都可以从属于这三大模块，这些功能之间的从属关系大致可以如下定义。

（1）微链核心

1）命令交互系统，用于与节点核心进行功能调用；

2）节点 RPC 服务，支持外部程序通过 RPC 的方式访问节点；

3）节点数据监听，用于与其他节点进行区块数据同步以及其他数据交换；

4）脚本系统，通过脚本的锁定与解锁执行交易合约；

5）区块链账本维护，用于验证网络中的区块数据并打包到主链。

（2）钱包功能

1）密钥维护，用于维护用户的公钥私钥和钱包地址；

2）发起交易，发起的交易需要被节点打包到主链才有效；

3）账务查询，如余额以及交易历史等。

（3）挖矿

1）区块生产，用于打包新的区块数据到主链；

2）货币发行，用于获取新的货币奖励以实现货币发行。

大家可以看到，在这里将微链的功能做了一个模块分类，然后实际实现一个区块链应用的时候，并不是说一定要去分别实现三个不同的独立程序，这只是一个逻辑上的分类，比如比特币就是将挖矿程序独立出去，在比特币的核心客户端功能中没有挖矿功能，然而以太坊却在核心客户端中集成了所有的模块，甚至还包含了一个合约程序编写与编译的功能（在不同的以太坊客户端中有差别）。所以，在理解一个区块链应用的结构时，不必过于严格僵化。

这里有几点需要提醒一下：

1）区块链应用程序显然是一个网络软件，否则也不存在什么达成网络共识之类的说法了，但是，注意了，一个区块链应用是可以单机运行的，甚至不需要联网。让我们看一段对话吧。

Alice：不是所有的区块链程序都是联网使用的吗？

Bob：站在实用角度来讲这是对的，比如比特币有自己的比特币网络，以太坊有自己的以太坊网络，如果没有这些网络，对于大家来说也就没有意义了，但是站在技术角度，它的运行并不一定要联网，你完全可以在自己的电脑上单机运行一个比特币程序，你的电脑也不需要联网，这并没有问题。

Alice：……

Bob：有点晕是不是，这么说吧，首先，每一个区块链核心节点程序都是独立工作的，在技术上，节点之间并不需要互相去调用对方的功能，也不存在要去访问某个服务器的做法，这与我们通常使用的支付宝、微信、网络游戏等是完全不一样的，比如我们使用支付宝转账，如果支付宝的服务器关闭了或者正好某一段网络断开了，那我们是没法自己独立使用支付宝的，甚至都无法登录。

Alice：这么说，我自己电脑上就独立安装了一个比特币程序，也不需要联网，然后就自己挖矿吗？

Bob：技术上可以这么做，程序也能不出错地运行，只不过我这里只是说在技术上可以这么做，作为一个软件程序，它可以正常运行。实际上，如果不联网，那么你的节点就是一个孤立的节点，比特币主网络中的数据你收不到，你自己的交易数据别人也收不到，从而相当于与比特币的主网络断开了，你的任何数据变更都是没有被主网络承认的。

Alice：我明白了，区块链的节点程序都是可以各自独立运行的，而且也都有各自独立的副本数据，只不过在运行的过程中，数据的变更需要在网络上广播出来，让大家都来验证一下，认可一下，从而成为大家都承认的合法数据。

Bob：是的，不过有时候为了自己测试方便，可以搭建一个自己的网络，通过网络标识号来区分一下，这个也称为私有链。

2）对于一个区块链应用程序来说，挖矿并不一定是必需的，在分布式的异步网络环境下，挖矿程序主要作用是让各个节点的数据副本达到最终一致，同时通过给矿工奖励货币的方式来发行新的货币。换言之，如果在一个区块链程序的运行场景中，拥有良好的网络，有足够的性能可以做到实时同步一致，挖矿就不是必需的，甚至在某些场景中如果不需要使用数字货币，那么连发行货币的功能都是可以不需要的，比如超级账本的设计、一些商业环境下基于区块链的审计系统等。

3）区块链应用程序在本质上其实就是一个P2P的网络软件，它的数据存储格式是区块链的方式，不同的节点之间互相可以同步数据，通过一个算法作为大家共同遵循的规则来达成共识。

接下来，我们就要开始演示搭建微链。

8.4　源码解析

8.4.1　目录结构

我们来看一下微链程序的目录结构，如下所示：

- bin
- blockchain
- cmd
- encrypt
- gopkg.in
- miner
- rundata
- script
- test
- tinynet
- transaction
- utils
- utxo

1）cmd：这是程序的主程序目录，其中包含了入口 main 函数以及一个命令行接口环境，微链就在这里启动。

2）blockchain：区块链程序，其主要的数据结构就是区块，其中就定义了微链的区块结构。

3）encrypt：微链中使用 RSA 算法生成私钥公钥以及钱包地址，并使用 SHA256 算法对事务以及区块计算哈希值，这些算法都定义在此目录中。

4）transaction：比特币中将一次转账交易或者挖矿获得新币的动作都称为事务，微链中也一样，在这个目录中，定义了事务的数据结构。

5）script：区块链应用中一个非常典型的特点就是可编程合约，比特币中使用了一组锁定和解锁脚本来表明一笔比特币的所有权，以太坊扩展了这个脚本的能力，将其变成了图灵完备的合约编程，微链中模拟了比特币的一组指令，这些功能就定义在这个目录中。

6）utxo：这是比特币中发起的一个概念，叫未花费输出，微链中同样模拟了这个结构。

7）miner：这是挖矿的定义。

8）tinynet：定义了微链的网络接口，支持 RPC 网络访问以及数据监听，可以认为是微链的网络模块。通过这个模块，可以使外部程序访问微链的核心，也可以使不同的微链节点之间进行通信。

9）rundata：作为一个演示程序，没必要将产生的数据都写入到硬盘中，也可以记录在内存中，比如区块链账本数据、UTXO 数据等，以太坊的模拟程序 TestRPC 便是将数据都模拟在内存中产生的。

10）utils：定义了一些工具方法，比如堆栈操作等。

以上便是一个常见的区块链应用所具备的代码目录结构，通过目录结构的名称我们也能看到所具备的功能设定，当然这里每一个目录中的功能都是可以有不同的实现的，比如上述的 UTXO，在有些应用中并不使用这样的结构，而是使用了账户结构（以太坊与比特币的区别之一）。另外，通过目录可以看到，微链中集成了核心节点功能、钱包以及挖矿功能。大家在自己进行实验编程的时候，可以根据自己的理解设定工程目录。

总之，一个区块链应用程序无非就是那么几个模块，用一句话来说就是：这是一种软件，使用区块链的结构存储数据，可以通过钱包进行转账交易等合约性质的事务操作，发生的事务会广播到其他运行的节点，这些事务数据最终通过矿工以执行挖矿算法的方式获得打包权后存储到区块链结构的数据中。可以看到，除了特有的区块链数据结构以及挖矿机制，其他的都没什么，就是一个普通的软件而已。（当然，就这两项的发明也已经够天才了。）

8.4.2 代码之旅

本节通过对代码做功能分析，让大家了解一下一个最简单的区块链程序到底是怎么组成的，有兴趣的读者也可以使用自己熟悉的语言环境尝试编写。这是一个很有趣的过程，区块链应用本来就是充满着实验性的软件，用它可以实验各种各样的想法，本身通过一套软件系统来发行货币就已经是够不可思议的了，这完全打破了人们对于软件作用的认识。

1. 主程序

有过软件开发经验的读者肯定知道，再小的程序也会有一个入口的启动程序，也就是主程序，在主程序中会进行一些系统参数的初始化、命令解释器的设置和一些服务的启动，如下所示：

```
package main

import (
    "fmt"
    "os"
    "tinychain/tinynet"
    "tinychain/utils"
    "gopkg.in/urfave/cli.v1"
)
var tinyapp = cli.NewApp()

// 应用初始化
func init() {
```

```
	tinyapp.Name = "gtinychain"
	tinyapp.Description = "a tiny example of blockchain procedure"
	tinyapp.Version = clientRevision
	tinyapp.Author = " 白话区块链 "

	//设置子命令
	tinyapp.Commands = []cli.Command{
		initCommand,
		versionCommand,
	}
	//设置命令参数
	tinyapp.Flags = []cli.Flag{
		utils.DataDirFlag,
		utils.NetworkIdFlag,
		utils.RPCEnabledFlag,
		utils.RPCPortFlag,
		utils.ListenPortFlag,
	}

	tinyapp.Action = gtinychain
}

//启动命令行主程序
func gtinychain(ctx *cli.Context) error {

	//发现并连接其他节点
	tinynet.DiscoverNodes()

	//进行区块主链的数据同步
	go tinynet.SyncBlockchain()

	//启动 rpc 与数据监听服务
	go tinynet.StartRpcServer(utils.Rpcport)
	go tinynet.StartListenServer(utils.Listenport)

	//启动命令解释器
	DoCommandInterface()

	return nil
}

func main() {
	if err := tinyapp.Run(os.Args); err != nil {
		fmt.Fprintln(os.Stderr, err)

		os.Exit(1)
	}
}
```

上述便是主程序的示例代码，由于微链被设计为一个命令行程序，因此使用了一个命

令行程序开发框架以便于实现子命令以及命令参数等功能，代码中引用的 gopkg.in/urfave/cli.v1 就是一个 Go 语言实现的命令行程序开发框架，这是一个托管在 GitHub 上的开源框架，可以通过网址 https://github.com/urfave/cli 查看源码实现以及详细使用说明。

我们来看看在主程序中主要做了如下哪些事情。

1）加载支持命令及命令参数。

加载后，可以在命令行中执行微链支持的各种主程序指令。假设微链的主程序名是 gtinychain，则可以在命令行中通过 gtinychain version 输出微链的版本号；通过 gtinychain init 进行数据目录和创世区块的初始化；通过 gtinychain --datadir 指定微链的数据目录等，以下列出了部分支持的命令。

```
//输出版本号
gtinychain version

//默认以当前所在目录进行微链初始化
gtinychain init

//以当前目录为数据目录启动，这是默认参数
gtinychain --datadir "./"

//指定rpc服务和数据监听端口启动
gtinychain  --rpcport 52000 --port 62000

//根据ipc文件启动rpc命令控制台
gtinychain attach --ipcfile='ipc文件路径'
```

这里说明一下微链初始化的命令，由 gtinychain init 可以通过读取配置文件来初始化一个创世区块（区块链的第一个区块），并且可以自动创建出一个钱包地址作为测试使用，钱包地址中也可以初始化一个可用金额，一般在进行初始化的时候还可以指定一个目录，那就会用到 --datadir 参数，则命令变成了如下形式：

```
gtinychain init --datadir "./"
```

这个命令将当前目录作为数据目录进行初始化，我们看看初始化命令大致是怎么做的：

```
initCommand = cli.Command{
        Action: func(c *cli.Context) {

                //根据参数指定路径，创建Data与Keystore
                //utils.Datadir就是通过参数 --datadir传入的路径
                os.MkdirAll(utils.Datadir+"/Keystore", 0777)
                os.MkdirAll(utils.Datadir+"/Data", 0777)

                //在Keystore目录下创建公钥私钥文件
                encrypt.GenerateRSAKey()
                //加载公钥私钥
                rundata.SetPrvPubKey()
```

```
        //加载账户地址，实际就是对公钥的格式化处理
        rundata.Account = encrypt.GetWalletAddr()

        //读取datadir指定目录下的genesis文件建立创世区块
        genesisFile, _ := ioutil.ReadFile(utils.Datadir + "genesis.json")
        var gsf GenesisFile
        json.Unmarshal([]byte(string(genesisFile)), &gsf)
        blockchain.CreateGenesisBlock()
    },
```

这就是init命令的大致过程了，主要任务就是创建钥匙和数据文件夹以及创建创世区块。钥匙文件夹（Keystore目录）是专门用来存储创建的私钥（钱包地址信息）的，这个文件夹极其重要，一旦丢失，等于这个钱包地址中的资产就丢了，跟现实生活中丢了钱包是一个道理，所以必须备份好。

注意这里的“加载公钥私钥”和“加载账户地址”，通常的区块链应用中并不必需有这么一个加载的动作，这是为了测试方便取数，所以将这些信息加载到内存中了（以太坊就有一个模拟测试程序叫TestRPC，是在内存中模拟加载整个环境的，目的还是为了便于测试）。实际上这里的公钥私钥创建是属于钱包的功能，这部分的功能在初始化过程中是可有可无的，如果在初始化的时候不创建，则可以在节点启动后通过命令另行创建。

2）区块链数据同步。通常一个区块链应用在初始运行的时候，或者说启动的时候，都会干一件事情，那就是区块链数据的同步（当然，这里指的是核心节点，如果只是使用独立的钱包功能或者挖矿程序，则其本身并没有同步完整区块链数据的需求），所有的操作都应该要等数据同步完成后才能进行，这是通过两个步骤来完成的：一个是发现其他节点，一个就是从其他节点获取数据。发现其他节点的方法有很多种，比如通过设计一个“发现协议”以广播的形式寻找同伴，类似于大家约定一个暗号，简单点的做法可以将其他节点的地址信息直接加载进来（类似于比特币的种子节点或者以太坊的星火节点），联系上其他节点后，就可以要求其他节点发送数据给自己了，其实就是一个下载的过程，只不过可以从多个联系上的节点那里同时下载，下载完成后，自己的节点就拥有了与网络中的主链一致的区块数据。

3）启动服务、RPC服务和数据监听服务。在Go中可以分别使用net/rpc以及net包来实现，就是一个网络监听服务而已。为什么这里要搞成两种网络监听服务呢？主要还是对比特币的一个模拟，在比特币中，如果使用外部命令或程序访问核心客户端，只能通过RPC的方式，并且与核心客户端要在同一机器上，也就是说禁止以远程的方式直接访问比特币的核心客户端，这是一个安全性的考虑。如果是核心客户端之间或者说是节点之间进行区块数据同步、数据交换等，则使用专门的数据监听服务。这两者的网络端口也是不一样的。微链在这里只是一个模拟，读者自己在尝试的时候，可以根据需要来决定。

RPC的小知识

RPC也就是Remote Procedure Call，远程过程调用的意思，它是一种基于网络的远程

功能调用协议，比如我们打开一款天气预报的手机App，软件向服务器发送一个获取天气情况的功能请求，远端的服务器收到请求后获取数据，再将结果响应给手机App，这就是完成了一次RPC。注意，RPC只是一种协议规范，不是一个具体的程序实现，这是一个比较泛的概念，因此有多种实现方式，比如数据的传输方式可以承载在HTTP或者TCP等协议上，而数据的编码可以采用json、xml、protobuf等格式。各种组合也各有优劣，这里不再赘述。

数据监听服务的实现在下面章节中有专门介绍，我们先来看一下RPC服务的示例代码：

```
package main

import (
    "net"
    "net/rpc"
    "net/rpc/jsonrpc"
    "tinychain/blockchain"
    "tinychain/transaction"
)

type Account int
type Block int
type Tnc int
type Miner int

///<summary>
///根据地址账号获得余额
///</summary>
///<param name="Account">地址账号</param>
///<param name="RemainAmount">返回余额</param>
func (ac *Account) GetBalance(Account string, RemainAmount *int) error {
    return nil
}

///<summary>
///根据区块号获得区块信息
///</summary>
///<param name="BlockNumber">区块号</param>
///<param name="BlockInfo">返回区块信息</param>
func (ac *Block) GetBlockInfo(BlockNumber int, BlockInfo *blockchain.BlockInfo)
error {
    return nil
}

///<summary>
///发送交易事务
///</summary>
///<param name="TransactionInfo">构造交易事务</param>
```

```
    ///<param name="Result">返回执行结果</param>
    func (ac *Tnc) SendTransaction(TransactionInfo transaction.TransactionInfo,
Result *int) error {
        return nil
    }

    ///<summary>
    ///关闭节点服务
    ///</summary>
    ///<param name="Signal">关闭信号</param>
    ///<param name="Result">返回执行结果</param>
    func (ac *Tnc) Close(Signal int, Result *int) error {
        return nil
    }

    ///<summary>
    ///开启挖矿
    ///</summary>
    ///<param name="Signal">开启信号</param>
    ///<param name="Result">返回执行结果</param>
    func (ac *Miner) Start(Signal int, Result *int) error {
        return nil
    }

    ///<summary>
    ///关闭挖矿
    ///</summary>
    ///<param name="Signal">关闭信号</param>
    ///<param name="Result">返回执行结果</param>
    func (ac *Miner) Stop(Signal int, Result *int) error {
        return nil
    }

    func StartRpcServer(port int) {

        lsn, _ := net.Listen("tcp", ":"+strconv.Itoa(port))
        defer lsn.Close()
        srv := rpc.NewServer()
        srv.RegisterName("Account", new(Account))
        srv.RegisterName("Block", new(Block))
        srv.RegisterName("Tnc", new(Tnc))
        srv.RegisterName("Miner", new(Miner))

        for {
            conn, _ := lsn.Accept()

            go srv.ServeCodec(jsonrpc.NewServerCodec(conn))
        }

    }
```

可以看到，在RPC服务中内置了一组支持的命令方法，比如获取区块信息、启动挖矿等，通过内置的命令解释器（下面会介绍）可以直接进行调用访问，也可以单独再提供一个客户端程序，通过RPC的方式连接访问，上述示例代码演示的是Go语言中的RPC编写方法，限于篇幅没有再给出每个方法的详细实现，读者了解是什么意思即可。在Go中，调用RPC服务也很简单，下面给出一个示例：

```
import (
    "fmt"
    "net/rpc/jsonrpc"
    "strconv"
)

func ClientForBalance(port int) {
    client, _ := jsonrpc.Dial("tcp", "127.0.0.1:"+strconv.Itoa(port))

    var targetAccount = "MIGfMA0GCSqGSIb3DQEB"
    var replyAmount int
    client.Call("Account.GetBalance", targetAccount, &replyAmount)

    fmt.Printf(strconv.Itoa(replyAmount))

}
```

通过json-rpc的调用即可实现与RPC服务的交互，上述代码演示了对获取账户地址余额的调用。通过上述的展示，我们可以看到，虽然区块链应用是一个个独立的客户端程序，运行过程中不需要专门连接一个服务器，但是其本身却集成了服务端功能，可以供外部访问调用。在现实世界中，比特币、以太坊等区块链应用都集成了类似的服务端，那些运行着的节点，其实就是服务器。

4）启动一个命令解释器，可以输入微链支持的命令与核心进行交互，可以看一下命令解释器的实现代码：

```
func DoCommandInterface() {

    client, _ := jsonrpc.Dial("tcp", "127.0.0.1:"+strconv.Itoa(port))
    defer client.Close()
    var cmd string

    for {
        fmt.Print(">>")
        fmt.Scanln(&cmd)

        if cmd == "exit" {

            //退出命令控制台
            os.Exit(1)
```

```
            fmt.Println("\n")

        } else if cmd == "tnc.getbalance()" {

            // 获得当前账号的余额
            var targetAccount string
            var replyAmount int
            fmt.Print(" 请输入账号地址 :")
            fmt.Scanln(&targetAccount)

            client.Call("Account.GetBalance", targetAccount, &replyAmount)

            fmt.Print(strconv.Itoa(replyAmount))

        }


    }
}
```

实际上连接 RPC 服务后，进入一个死循环，然后接受各种支持的字符串指令，这里演示了两个功能：第一个是退出命令解释器，第二个是调取某个账号地址的当前余额。显然根据 RPC 服务支持的功能，支持的命令还远不止这些，但是原理都是一样的，常用的命令如下：

- admin.close()：关闭连接的节点服务，注意不是关闭 RPC 服务而是关闭整个节点实例的运行。
- tnc.getblocknumber()：获得当前最新的区块号。
- tnc.getbalance()：获得当前账号的余额。
- tnc.sendtransaction()：发送一笔转账交易。
- miner.start()：开启挖矿。
- miner.stop()：停止挖矿。

实际支持的命令可以根据需要去拓展，这里只是一个代码样式的演示，这些命令都是通过微链的节点核心来执行的。其他的区块链应用也基本都会提供这样的访问接口，在比特币中，可以通过图形界面的客户端程序进行命令交互的调用，也可以通过一个独立的命令行程序来访问，在以太坊中，则可以在节点程序启动的时候同时启动一个交互式的控制台来访问节点。

再介绍一个小功能，有时候我们可能希望同时开启多个命令控制台，比如在 1 号控制台运行挖矿指令，在 2 号控制台运行区块查询指令，就像我们日常工作时，经常会为电脑连接多个显示器一样。要实现这个功能很简单，只要在启动第一个命令控制台的时候，在某个目录下生成一个文本文件，可以命名为 tiynchain.ipc 或者任何其他的名字，文件中存储 RPC 服务的连接地址即可，然后通过 gtinychain attach --ipcfile=' 文件路径 ' 这样的命令

来启动一个新的命令控制台。

2. 区块的定义

作为一个区块链应用程序，其核心的数据结构就是区块了。一般来说，区块中包含的信息主要分为区块头和区块体，区块头中包含区块的摘要信息，区块体中包含区块事务。至于区块事务是指什么，取决于不同的应用程序，比如比特币中主要就是交易信息，从一个地址到另外一个地址的交易记录。微链也同样模拟了这一点，我们来看一下区块的定义：

```
type BlockInfo struct {

    // 区块编号
    blockNumber int

    // 前一个区块哈希
    hashPrevBlock string

    // 交易事务的 merkle 根
    hashMerkleRoot string
    // 区块打包的时间戳
    nTime uint32
    // 难度位数
    nBits uint32
    // 随机目标值
    nNonce uint32

    // 交易事务
    trans []transaction.TransactionInfo
}
```

可以看到，在这个区块中同时包含了摘要信息和区块交易事务。在摘要信息中，当前区块通过“前一个区块哈希”与之前的区块连接，这也是区块链名词的来源。其中的区块编号就是区块的高度，一个一个区块通过区块哈希连接起来后，每增加一个新的区块高度就增加 1，当我们需要查询某个区块的信息时，可以提供一个方法传入区块编号输出区块的说明信息。交易事务的 merkle 根是对区块中所有的交易事务进行哈希计算，构造一棵 merkle 树而得来，可以用来验证区块的完整性。难度位数和随机目标值是与挖矿有关的参数。

在大多数的区块链应用中，区块的结构定义基本上都采用了上述方案。这里的交易事务其实也不仅仅只能用来表示交易。微链中只支持数字货币的转账交易，但是在很多功能比较强大的区块链应用（如以太坊）中，事务的概念是更加宽泛的，除了表示转账交易外，也可以表示某个状态的变更，比如多重签名、合约有效期变更等。

上述代码中提及的哈希算法是 SHA256 算法，对于实验代码来说，使用何种哈希算法

是没有要求的，如果是正式版的程序，要使用比较强壮的抗碰撞能力强的算法。看下微链中的示例代码：

```
//sha256
func GetSHA256(msg string) string {
    hData := sha256.New()
    hData.Write([]byte(msg))
    return fmt.Sprintf("%x", hData.Sum(nil))
}
```

这是一个很简单的使用。

3. 事务的定义

微链中参照了比特币的事务结构，使用了输入和输出的方式来表示一个交易事务：

```
type OutPoint struct {
    //事务哈希
    tranHash string
    //事务的输出部分的索引号
    n int
}

type TxIn struct {
    //指向前一次的输出
    prevOut     OutPoint
    //前一次的输出索引
    sequence    int
    //解锁脚本
    scriptSign script.ScriptAction
}

type TxOut struct {
    //金额
    amount          int
    //锁定脚本
    scriptPubKey script.ScriptAction
}

type TransactionInfo struct {
    //交易事务哈希
    tranHash string
    //输入集合
    txIn []TxIn
    //输出集合
    txOut []TxOut
    //时间戳
    lockTime int64
}
```

上述代码示例中，TransactionInfo 就是事务的结构定义，一条事务在这里可以理解为一

笔交易，每一笔交易都有自己的哈希值，就像身份证号一样，唯一地表示了某一笔发生的交易。我们日常在进行银行转账的时候，通常会先往自己的账户里存钱，然后再转出到目标账户。换句话说，就是有一个存入和存出，在这里也是一样的，事务的结构中，txin 表示存入或者说来源，txout 表示输出。如果账户里本来就有钱，不要先存入再转出，可以直接就转出，那这里的输入还需要吗？回答是：需要，而且必须要。因为在微链中并不会把存入的金额记下来，而只会记录每一笔进账和出账的流水账，因此每一笔的输出都要指定它的输入来源，这样才能保证账是平的。

对于输出，很好理解，我们可以看到 TxOut 的结构定义，就是一个金额然后一个锁定脚本（一段指令程序），关于脚本我们下一节再解释，这里可以理解为一个标记，标记着这笔交易的接收方，接收方可以使用解锁脚本（也是一段指令程序）来使用这笔发给自己的金额。

对于输入，实际上就是指向之前其他事务对自己的输出，比如别人之前对我有一笔 100 的输出，现在我把这个输出作为输入，输出或者说转给另外一个人。因此我们看到在输入 TxIn 的定义中，主要定义了指向前一次的输出，然后就是一个解锁脚本。

我们发现，在微链的定义中，除了区块是一个个串接起来的，交易事务也是串接起来的。要构造一个交易事务，其实就是构造事务的输入和输出，下面简单演示一个挖矿的交易（没错，矿工挖矿的收入所得也属于一种交易，为了区分普通的转账交易，这种交易通常称为 coinbase 交易）。

```
    //coinbase 事务的输入部分
    var txIn []TxIn
    txIn = make([]TxIn, 1)
    txIn[0].prevOut = OutPoint{tranHash: "", n: 0}
    txIn[0].scriptSign = script.ScriptAction{InSignData: "", InPubKey: "", OutPubkey:
""}
    txIn[0].sequence = 0

    //coinbase 事务的输出部分
    var txOut []TxOut
    txOut = make([]TxOut, 1)
    txOut[0].amount = targetAmount
    txOut[0].scriptPubKey = script.ScriptAction{InSignData: "", InPubKey: "",
OutPubkey: targetPubKey}

    //事务时间戳
    curTime := time.Now()
    timestamp := curTime.UnixNano() / 1000000

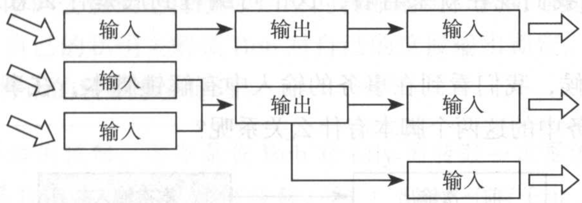
    //事务哈希
    var tranHashCoinbase string
    //通过一个方法计算出整条事务的哈希值
    tranHashCoinbase = GetTransactionHash(txIn, txOut, timestamp)
```

```
    tranInfo.lockTime = timestamp
    tranInfo.tranHash = tranHashCoinbase
    tranInfo.txIn = txIn
    tranInfo.txOut = txOut

    return tranInfo, nil
```

在 coinbase 交易中，输入方比较特殊，并不是来自之前的输出，而是由系统通过奖励的方式直接发行出来的。注意，若是一个矿工挖到了矿（获得了区块打包权），在打包一个区块中的交易时，通常会把属于自己的 coinbase 交易放到区块中所有交易的第一位。打包完成后会将区块信息广播出去，等待其他节点来进行数据校验以及同步，当大多数节点校验通过后，这笔交易就算是被网络认可了。

每进行一次事务交易，就会产生一个新的输出，这些新的输出都是属于某个地址的可花费输出，当某个地址的所有者需要向其他人转账交易的时候，可以创建一笔新的输入和输出，这个新的输入就是来自自己的可花费输出。输入输出的关系如下图所示：



可以看到，事务的输入输出中，是彼此衔接的关系，生活中有很多这样的例子，比如仓库中的入库和出库，银行账户中的存款和取款，通过这样的流转实现了价值的转移，因此可以说带有金融属性的区块链网络是一个可以实现价值传输的网络，而且还是去中心化的。

为了方便查阅某个账户地址下的可花费输出，也就是 UTXO（Unspent Transaction Output），通常会单独设置一个独立的 UTXO 数据存储，比如：

```
//未花费输出
type UTXO struct {
    TranHash     string
    Sequence     int
    Amount       int
    ScriptPubKey script.ScriptAction
}

//账户的UTXO
var TinyUTXO []UTXO
```

未花费输出的结构与事务中的输出其实是一致的，只不过这里多了一个“未花费”的条件约束，相当于净值。每当需要对别人进行转账的时候，可以直接到属于自己的“未花费输出”中去搜索指定，如果想知道自己的地址下一共有多少余额，也可以通过“未花费

输出”来获取：

```
func GetRemainAmount() int {

    var rAmount int
    for _, v := range TinyUTXO {

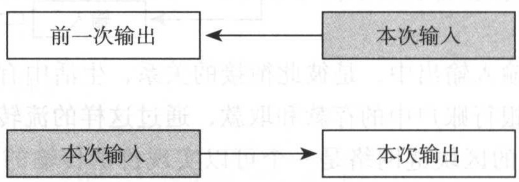
        rAmount = rAmount + v.Amount
    }
    return rAmount
}
```

总而言之，UTXO 的存在，是为了方便计算某个地址下的可用输出（余额）。

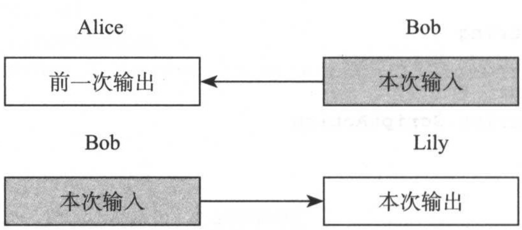
4. 脚本的定义

脚本可以说是区块链应用中一个极其重要的特性，我们经常说基于区块链的各种数字货币都是可编程货币，基于区块链的各种合约也是可编程合约，这种特性开启了可编程社会的一个新的起点。我们现在就来看看，这个可编程到底是什么意思，它大致是怎么实现的。

在介绍事务的时候，我们看到在事务的输入中有解锁脚本，在事务的输出中有锁定脚本，那么，在一个事务中的这两个脚本有什么关系呢？

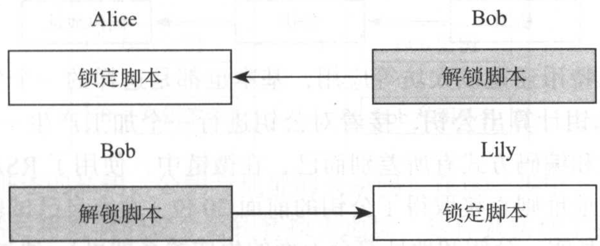


这是一个简单的示意图，大家注意其中的“本次输入”，在一次交易事务中，是由一组“本次输入”和“本次输出”组成，而“本次输入”又是来自“前一次输出”，因此对应“本次输出”的真正来源其实是“前一次输出”，这么说也许还有些抽象，让我们给这些动作赋予一些角色吧。



这里我们假设 Bob 本来是一无所有的，他之所以能够转账输出给 Lily，是因为之前 Alice 转了一笔钱给他。Alice 转账给 Bob 的时候，通过锁定脚本标识了这笔钱的所有权，而这个锁定的标识只有 Bob 通过自己的钥匙才能解开，从而能够使用 Alice 给他的这笔钱，Bob 用来解开这个标识所使用的工具就是解锁脚本。Bob 解锁了 Alice 给他的输出后，就可

以自由地使用 Alice 转给他的钱了，那么让我们站在解锁与锁定的角度再来看一下这幅图：



这里的锁定与解锁脚本其实就是一段验证程序，原理跟古时候的军队虎符差不多，在微链中，是通过使用公钥算法来实现这种虎符机制的，公钥算法的原理在这里不再赘述了，在之前的章节中已经有了叙述。Alice 使用 Bob 的公钥锁定了自己对 Bob 的转账输出，这段锁定程序就是 Alice 对 Bob 的输出锁定脚本。Bob 在对 Lily 转账的时候，首先解锁了 Alice 对自己的那笔输出，通过自己的私钥解锁了 Alice 对自己的那段锁定程序，相当于虎符匹配上了，然后再使用 Lily 的公钥锁定自己对 Lily 的转账输出，从而等到 Lily 想要使用这笔钱的时候，得使用 Lily 自己的私钥去解锁 Bob 对自己的这段输出锁定。看看，多么环环相扣的设计啊，这也是比特币的转账事务原理。

注意　上述只是一个举例说明，并不是说 Bob 对 Lily 的转账一定要使用 Alice 对 Bob 的那笔输出，如果 Bob 有很多人对他转账，除了 Alice 还有 Eric、Gerge 等，那么 Bob 是可以任意选择要花哪一笔的（反正都是自己的钱）。

让我们看下微链中的脚本定义吧！

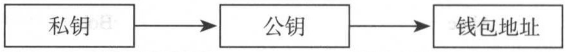
```
type ScriptAction struct {
    //解锁脚本中的私钥签名
    InSignData string

    //解锁脚本中的公钥
    InPubKey string

    //锁定脚本中的公钥
    OutPubKey string
}
```

还是以上述的 Bob 为例，在 Bob 要转账给 Lily 的时候，Bob 提供了 Alice 对自己那笔输出的解锁脚本，解锁脚本中包含了自己的私钥签名和公钥，也就是在此时，脚本中的 InsignData 和 InPubKey 都是 Bob 提供的，然后在构建对 Lily 的输出的时候，使用了 Lily 的公钥来锁定（因为 Lily 的公钥只有 Lily 使用自己的私钥才能解密，从而也就保证了这笔账款的输出只有 Lily 的私钥拥有者才能解锁），也就是此时的 OutPutKey 是指 Lily 的公钥数据。

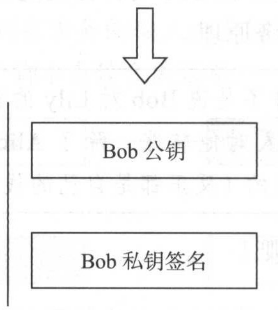
在微链中，生成一个用户的钱包地址时，过程如下：



实际上无论是比特币还是以太坊等应用，基本也都是这样的一个生成关系，先生成一个私钥，然后通过私钥计算出公钥，接着对公钥进行一个加工产生一个所谓的钱包地址，只是各自的算法方式和编码方式有所差别而已，在微链中，使用了 RSA 算法作为私钥公钥的生成算法，而钱包地址则直接取得了公钥的前面 20 位（大家自己试验的时候可以根据自己的设计来，总之让私钥、公钥和地址符合上面的生成关系即可），通过这些我们也能看到，实际上并不存在一个真正的所谓的钱包地址，只有私钥和公钥，地址只是公钥经过某种转化的数据而已。

我们看一下发起一个交易的时候，锁定和解锁脚本是怎么工作的？我们设定一个场景，假设之前 Alice 转账 100 给了 Bob，现在 Bob 要将这 100 转账给 Lily。

1）第 1 步自然是要构造一个事务，即如之前所述，Bob 要转账给 Lily，首先得把 Alice 给自己的转账输出解锁，看一下如下过程：



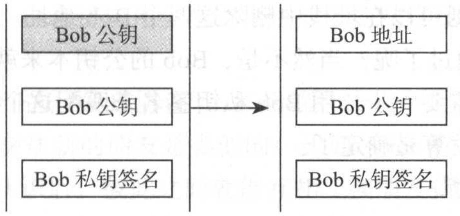
2）Bob 的解锁脚本将自己的私钥签名和公钥压入了一个堆栈，这个是用来解锁 Alice 给自己的转账输出的，看一下代码示例。

```
//压栈方法
func (sa *ScriptAction) OP_PUSH(anyData interface{}) {
    myStack.Push(anyData.(string))
}

//压入 Bob 的私钥签名和公钥到一个堆栈中
OP_PUSH(BobSign)
OP_PUSH(BobPubKey)
```

堆栈的定义这里就不再赘述了，总之就是一个后进先出的存储结构，这个步骤相当于 Bob 亮出了自己的身份证，接下来就希望 Alice 给自己的那段输出锁定来验证自己的身份。

3）Alice 给 Bob 的输出脚本中包含了 Bob 的地址，这个地址要与 Bob 的地址进行匹配，看看是否一致，怎么处理呢？首先在堆栈中要给出 Bob 的地址，然后将 Alice 输出脚本中包含的 Bob 的地址也压入堆栈，通过一个方法来判断是否一致。

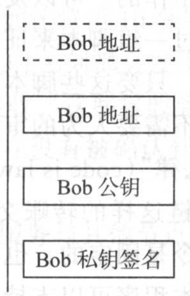


4）还记得上面讲述的私钥、公钥与地址的关系吗？在这一步中，首先在堆栈中复制了一个 Bob 的公钥，然后将复制的这个公钥转换为地址，这样就实现了在堆栈中给出 Bob 的地址，可以看下代码示例：

```
// 将栈顶的公钥数据取出后取得前面 20 位
// 这 20 位作为钱包地址，并且再次压入堆栈
func (sa *ScriptAction) OP_PUB20() {
    var pubKey = myStack.Pop().(string)
    var s = []rune(pubKey)
    var bfr20 = s[0:19]
    myStack.Push(bfr20)
}
```

代码很简单，就是将公钥转换为地址，各个不同的区块链应用有不同的转换方式，这里就取得公钥的前 20 位作为一个例子（比起比特币中的方法可是简陋多了）。

接下来将 Alice 输出脚本中包含的 Bob 的地址也压入堆栈。



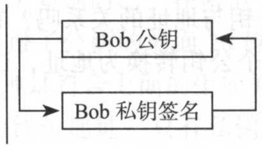
以下代码演示了从堆栈中取出两个数据，并比较是否相等的过程：

```
// 从堆栈中取出两个数据，比较是否相等
func (sa *ScriptAction) OP_EQUALVERIFY() bool {
        //myStack.Pop() 会从堆栈中取出栈顶数据后，然后在堆栈中删除掉
    fstData := myStack.Pop().(string)
    sndData := myStack.Pop().(string)
    return strings.EqualFold(fstData, sndData)
}
```

虚线标记的 Bob 地址就是来自 Alice 对 Bob 的那笔输出脚本，现在就可以来验证一下 Bob 的身份了，比较堆栈中的两个 Bob 地址是否一致。如果不一样那就有问题了，说明这个 Bob 可能是一个假的 Bob，如果是一致的，说明 Bob 提供的公钥和 Alice 提供的输出中

的Bob的地址是匹配的，则可以在堆栈中删除这两个Bob地址。那么，接下来是否就表明Bob的身份已经完全认证通过了呢？当然不是，Bob的公钥本来就是公开的，谁都可以用他的公钥来做匹配，因此还需要一步，用Bob私钥签名来匹配这个Bob公钥，如果这一步也是一致的，那Bob的身份就算是确定了。

5）Bob的公钥私钥匹配。



如图，Bob提供的私钥签名与Bob的公钥进行匹配，匹配成功后，Bob就可以创建针对Lily的输出了，同样，Bob使用Lily的公钥创建了输出脚本（或锁定脚本），这样Lily想要使用这笔钱，就要上述同样的步骤来进行验证确认。

至此，就演示了微链中的锁定脚本以及解锁脚本的工作过程，注意这里的名词，一个事务中的输入脚本就是解锁脚本，而输出脚本就是锁定脚本。这个工作过程也是模拟的比特币，可以看到，公钥密码算法在这里起到了很大的作用，这也是为什么像比特币这样的数字货币被称为加密数字货币的原因。

5. 关于脚本的一点思考

上述几节演示了脚本系统是怎么工作的。可以发现，在微链中，每个地址所拥有的货币并不是存储在一个账户的，而是通过一组脚本来证明所有权，不断转账的过程其实就是不断进行脚本的解锁和产生新的锁定，只要这些脚本程序一直能正常运行，这种转换就能依赖脚本程序生生不息地运转下去，不需要人为的审核，不需要查看身份证，一切都遵循着既定的规则，这其实就是“代码即法律”（code is law）的思想，是不是很酷！

那么，这些脚本除了能够用于微链这样的转账交易，还能干些什么？答案是肯定的，微链是模拟了比特币的做法，脚本指令是固定的，也因此只能干些交易转账的事情，如果对脚本的功能进行扩展呢？比如让脚本程序可以支持更多的操作，实现更丰富的功能，有无可能？确实是可以的，以太坊就是在比特币的基础上，大大增强了脚本的能力，不但实现了比特币的所有功能，而且还可以让用户自定义脚本，使用以太坊支持的脚本语言进行自定义编程，实现如自定义代币（在以太坊中使用脚本程序创造自己的数字货币）、众筹合约、自治组织等各种丰富的应用程序。

以太坊通过扩展增强脚本能力，实现了除了数字货币以外的其他合约功能，也称为智能合约。事实上数字加密货币本身就可以看作一种合约，合约的有效条件就是解锁脚本与锁定脚本进行匹配。

6. 网络服务

一个区块链应用如果不具备网络服务功能，那么就只能是一个单机的测试程序，没有

任何实用价值。一般来说，区块链应用至少要具备如下的网络服务功能。

（1）节点发现

通常可以设计一个专门的“发现协议”，目的就是让一个个独立运行的节点之间能够互相联系，这个与我们平时交往新的朋友是类似的。比如 Alice 有 10 个好朋友，Bob 有 15 个好朋友，当 Alice 与 Bob 认识后，彼此之间就能互相交换朋友信息，这样 Alice 和 Bob 就分别认识了更多的朋友，然后这些朋友之间还能彼此再认识，通过这样的方式每个人都会认识越来越多的朋友。有时候为了更加方便大家去尽快认识新朋友，还会设置一些种子节点，这些节点会不间断地长期运行着，刚刚加入的新节点可以首先去认识它们，相当于带个路。

（2）区块主链同步

由于每一个节点都是独立运行的，大家并没有一个统一的服务器作为同步参照，因此只能靠互相之间进行数据同步，比如 Alice 的节点目前的区块长度是 10，通过网络监听发现目前网络中最新的主链长度已经是 11 了，则 Alice 节点就会问身边的朋友要数据，大家都会彼此帮忙。

（3）新区块验证

当有矿工打包出了一个新的区块后，就会将区块数据广播出去，以尽可能地让更多的其他朋友获知，每一个节点都会敞开大门接收新的区块数据，接收到后就会进行自己的一轮验证，通过后就放到自己的仓库中（区块主链账本）。

（4）内存池维护

区块的生成是有时间间隔的，比如比特币 10 分钟一个区块，以太坊是 15 秒一个区块，但是交易并不是间隔发生的，而是无时无刻都会发生。某个节点上发生了一笔交易后就会立即广播出去，其他的节点会负责接收，这些接收到的交易事务都是需要等待验证以及被打包到区块的。这里面会有时间差，在没有被确认到区块主链之前，就会先保持在内存池，相当于一个临时储藏室。

在微链中，会启动一个数据监听服务与其他节点进行联络，我们看一下代码示例：

```
//启动监听服务
func StartListenServer(port int) {

    listenSocket, err := net.ListenUDP("udp4", &net.UDPAddr{
        IP:   net.IPv4(127, 0, 0, 1),
        Port: port,
    })
    if err != nil {
        fmt.Println("监听服务启动出错：" + err.Error())
    }
    fmt.Println("监听服务正在运行中...")
    defer listenSocket.Close()
    for {
        handleNodeMessage(listenSocket)
    }
```

```
}

var nCount = 0
// 消息处理方法
func handleNodeMessage(conn *net.UDPConn) {
    defer conn.Close()
    var bufferData [1024]byte
    var recCommand string

    // 获取接收到的数据
    for {
        n, clientAddr, err := conn.ReadFromUDP(bufferData[0:])
        if err != nil {
            fmt.Println("handle message error:" + err.Error())
        }

        // 将监听到的数据指令放到一个字符串中
        recCommand = string(bufferData[0:n])
        // 调用检测方法，确保获得的是一个合法的指令
        if CheckCommand(recCommand) == false {
            continue
        }
        nCount++
        // 首次连接时发送一个欢迎词
        if nCount == 1 {
            conn.WriteToUDP([]byte("欢迎访问！"), clientAddr)
        } else {
            switch recCommand {
            case "syncblock":
                fmt.Println("区块数据同步请求")
                break
            case "transbroad":
                fmt.Println("交易事务广播")
                break
            case "nodeexchange":
                fmt.Println("节点信息交换")
                break
            default:
            fmt.Println("other")

            }

        }

    }
}
```

```
// 指令格式校验
func CheckCommand(s string) bool {

    // 一系列的命令格式校验
    return true
}
```

这里给出了一段监听服务的代码示例，可以看到其实就是一段普通的 UDP 服务，微链可以通过这个服务监听其他节点发送过来的数据同步请求以及要求交换节点网络地址和端口的信息等，这是节点与节点之间的网络通道。

7. 挖矿

挖矿的目的是为了维持各个节点之间数据的共识，矿工（运行挖矿程序的计算机）通过执行运算挖矿程序抢夺到区块数据的打包权，打包后将产生新区块的信息广播到其他节点，并同步给其他节点。而系统也通过给矿工分配挖矿的奖励来发行新币，矿工为了得到新币的奖励就会持续运行挖矿程序，从而通过这种激励的方式维持了系统的运转。

挖矿程序通过运算一个什么样的程序来抢夺打包权呢？在区块结构的定义中，我们看到有一个难度位数和一个随机目标值，微链中借鉴了比特币中的挖矿算法，通过对一个难度值进行随机匹配来抢夺区块数据的打包权。举个例子，每一个区块都有一个难度目标值，比如第一个区块或者说创世区块的难度值是 0x000FFF，这是一个约定的数值，在微链中认为这个难度值的难度是 1。这其实就是玩一个游戏，比如我们掷骰子，要求连续掷 6 次，前 3 次必须都是 0，但是后面 3 次加起来的点数不能大于 18，我们在玩这个游戏的时候，就得要不断地掷骰子，大家一起比赛，看谁先抛出一个符合要求的点数出来。挖矿程序也是类似的原理，就是在不断地做这么一件事。

1）计算出当前区块的哈希值 H。注意，这个区块是指矿工整理好准备要打包的新区块；

2）在 H 后面附加一个随机数，然后连起来再做一次哈希计算，看得到的结果是不是符合要求。如果结果不符合要求就更换随机数来继续尝试；

3）如果在挖矿过程中收到了其他节点发送过来的新区块信息，表明当前高度的区块已经被确定了，矿已经被别人挖走了，这个时候就只好放弃了。继续下一个区块数据的计算。

在这种情况下，为了保持出块速度的均衡，每隔一段时间就需要调整一下难度，比如微链的出块速度大致维持在 30 秒，则可以设置每个星期调整一次难度，按照 30 秒来估计，一个星期大约会出 20 160 个区块，则系统设置为每间隔 20 160 个区块调整一次难度值，如此，则新的难度值 = 老的难度值 ×（最近 20 160 个区块的实际出块总秒数 /604 800），这里 604 800 表示理论上出 20 160 个区块的秒数，通过这样的公式计算均衡了一段时间产生的算力误差。

所谓的挖矿过程基本就是这样的，我们可以看到，为了得到符合要求的结果，就必须

找到那个随机数，就得不断重复尝试，这是多累人（不，是累CPU）的活啊，也难怪大家都称之为“挖矿”。当然，目前有不少区块链应用已经发展出了其他的挖矿算法，有些是不用消耗算力的，各种变种算法也是层出不穷。对于这部分的代码，读者可以根据自己的理解去做一个实现。

8. 钱包

钱包客户端在区块链应用中主要用来存储自己的私钥，通过私钥就能获取到自己的地址上有多少数字资产（不一定是数字货币，也可以是一个商业智能合约），也可以发起一笔转账交易或者创建一份合约等。事实上，钱包的功能并没有一个严格的规定，除了标准的私钥管理以及查询数字资产等，还可以将钱包的功能通用化，比如可以设计一个管理多种数字货币的功能，以便于用户管理自己的各类数字资产；还可以连接主要的交易平台以方便数字资产的交易，当然这个要与交易平台对接。总体来说，钱包的功能就是提供给用户一个区块链程序的使用工具。前面演示的发起一个交易事务，查询一个账户地址的余额等，实际上就是属于钱包的功能。

8.5 微链实验的注意问题

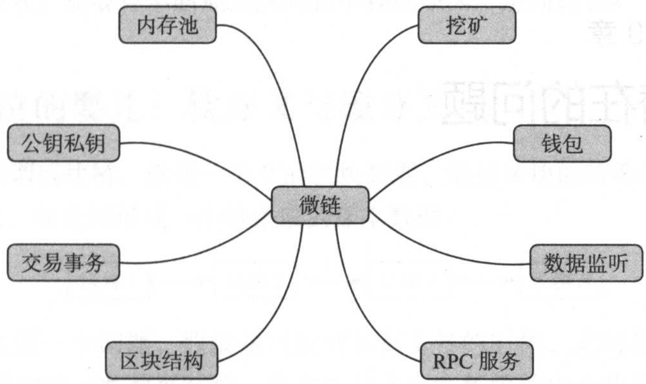
我们在实验开发微链的时候，为了减少复杂度，去除了相当多的异常处理，就以上述的微链设计来讲，是有很多问题没有考虑的，比如：数字货币仅支持整数货币；基本不做错误处理；区块数据维持在内存中；不支持创建多个地址；不考虑临时分叉的情况；出块的时间戳顺序校验；没有严格的区块数据验证；不支持数据的并发处理……

很多问题都是没有去细化的，这一点读者一定要注意。要开发一个真正能大规模使用的区块链应用，要考虑非常多的细节，任何一个问题的疏忽，都会留下潜在的巨大威胁，目前的知名公链系统（如比特币、以太坊等），在这些年的运行过程中都暴露过很多问题，直到现在也仍然有很多潜在的问题威胁。不过值得庆幸的是，作为开源软件项目，社区的力量是巨大的，无数多专业且热心的开发者不断提出各种改善方案，为系统的健康运转添砖加瓦。我们在本章通过微链的功能展示以及代码示例，可以基本了解一个区块链程序是如何编写的。说一千道一万不如去码一份实实在在的代码。有兴趣的读者可以根据自己的想象力，按照自己的想法设计一个有意思的区块链应用，从最简单的开始，逐步完善，为这个领域的发展贡献一份力量，便是再好不过了。

8.6 知识点导图

如果说区块链技术就像是一片星空等待我们去探索，那么本章所演示的仅仅只是一粒尘埃，无论是底层设施还是各种应用设计，都有着巨大的想象空间。区块链技术也并不只

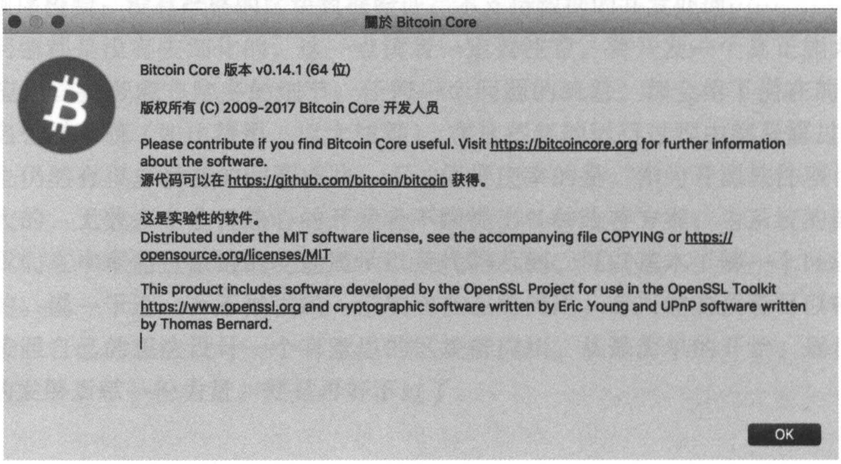
是独立的存在，其与传统的数据平台，或者说区块链的外部世界也在不断进行融合对接，比如将链外数据喂入到链内的预言机、见证人、数据审计技术，而不同的链之间也在进行多链的数据对接，这将是多么五彩缤纷的场景。我们就从最简单的微链起步，像滚雪球一样不断完善和积累，从而迈向未来。下面我们看一下本章的一个知识点思维导图：



Chapter 9 第 9 章

潜在的问题

任何一个软件系统都很难做到十全十美，在实际的使用过程中，会经受各种问题的考验。不同类型的软件都会有自己特有的问题，区块链应用作为一种具有特有功能的软件系统，也有着自己特有的问题。而本身区块链应用的设计思想就是一种实验，需要经过时间的考验。在全世界第一个区块链应用程序比特币的“关于”说明中，也指明了这是一类实验性的软件系统。



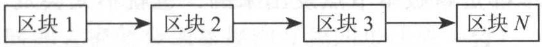
如图所示：“这是实验性的软件。”事实上从比特币开始，之后的所有区块链应用系统都是实验性的软件，链式的账本结构、去中心化的思想、最终一致性的特点等，这些设计特点能否在实际的商业应用中稳定运行，都需要大量的实验论证。

事实上，这些年各类区块链应用在使用过程中已经暴露了各种问题，甚至还发生过重

大的漏洞事故，区块链技术被认为可以很好地应用在金融、审计、支付、见证等领域，这些领域的软件系统必须是可靠的，因此在我们计划应用区块链技术来提供这些领域的服务时，必然要充分地了解可能会发生的各种问题。人类历史中，所有技术应用都会经历一个发展阶段，早年的火枪技术容易炸膛，手术技术容易感染，手机也只能用来打打电话。一个有发展前景的技术，需要在不断地发现问题中找到原因，进而改善。

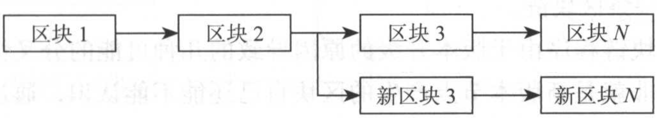
9.1 两个哭泣的婴儿：软分叉与硬分叉

我们知道，所谓区块链，就是一个个的区块数据，通过区块的哈希值（相当于区块的身份证号）串联起来，如此而形成一个链条般的账本数据。



在这里先问大家一个问题，假设在区块增长到 2 号的时候，此时软件升级了，增加了之前版本中不能识别的一些数据结构，会发生什么？在传统的中心化软件体系中，似乎并没有什么问题，无论是微信、支付宝、美团等，隔三差五就升个级，能有什么问题呢。

这是因为这些中心化的系统，数据存储都是集中的，版本管理也是集中的，如果是重大的升级，完全可以设置为若不更新到最新版就不能进行登录操作，从而确保用户使用的总是正确的版本。然而区块链先天是去中心的使用方式，一旦有新的软件版本发布后，是不是每个人都会去升级到新版本是很难控制的，这就可能导致如下图所示的问题。在 2 号区块生成的时候发布了新的版本，且新的版本增加了之前版本不能识别的数据结构，此时部分用户升级了新版，部分用户还没有升级，这些新旧版本的软件仍然在各自不停的挖矿、验证、打包区块，一段时间过后就会变成这样：



这个就叫分叉，现在读者应该很容易理解了吧，实际上根据不同情况，分叉的情况可以继续细分为如下两类。

1. 新版本节点认为老版本节点发出的区块 / 交易合法

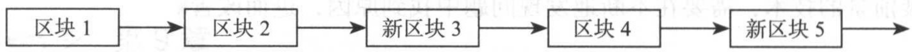
此时对于新版本来说，仍然是可以保留之前的区块链数据的，因为完全兼容，但是老版本的节点是否能依然接受新版本节点发出的区块就不一定了。

问：新版本能兼容老版本的区块，这个我能理解，但是老版本不一定是什么意思？难道说老版本还能继续识别新版本？新版本升级之后既然增加了新的数据结构，老版本肯定不能识别呀？

答：这个确实是需要分情况的，比如老版本中有一个备用的数据字段，这个数据字段

一直都是闲置的，在老版本中也没起什么作用，而新版本使用了这个备用的字段，此时由于老版本本来也没使用这个备用字段，因此对于新版本发出的区块是依然能接受的，相当于欺骗了老版本节点。

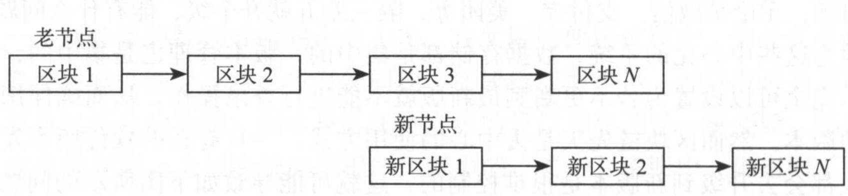
这种情况下，区块链的生成如下图所示：



可以看到，此时在区块链中，无论是老节点维护的区块链数据还是新节点维护的区块链数据，都有可能既包含老版本的区块也包含新版本的区块。不过实际上，在区块链应用程序进行重大升级时，都会事先取得社区的投票同意，保证大部分的运行节点都愿意升级到新版本，这种情况下，由于新版本节点的算力要大于老版本的节点，所以一旦完成升级后，后续的打包区块基本都是新版本节点发出来的，也就不太会发生老版本区块和新版本区块交错链接的情况。

2. 新版本节点认为老版本节点发出的区块 / 交易不合法

这种情况下，新版本节点基本上就是另外一套区块链程序了，如下图所示：



老节点如果还能接受新节点发出的区块，那么在老节点维护的区块链数据中，还有可能会插入新版本的区块，但是对于新节点来说，不再会有老版本的区块了。不但不接受新产生的老版本区块，对于之前的老版本区块也不再认可，因此这种情况下等于新版本的节点单独另外开辟了一条区块链。

上述解释了区块链程序由于版本升级的原因导致的几种可能的分叉情况，实际上站在老节点的角度，无非就是新版本节点产生的区块自己还能不能认识，通过能不能认识，导致两种类型的分叉：软分叉和硬分叉。

（1）软分叉

老节点不能发现新协议的变化，从而继续接受新节点用新协议所挖出的区块，这种情况称为软分叉，此时老节点矿工将可能在它们不能完全理解和验证的新区块上继续添加区块。

（2）硬分叉

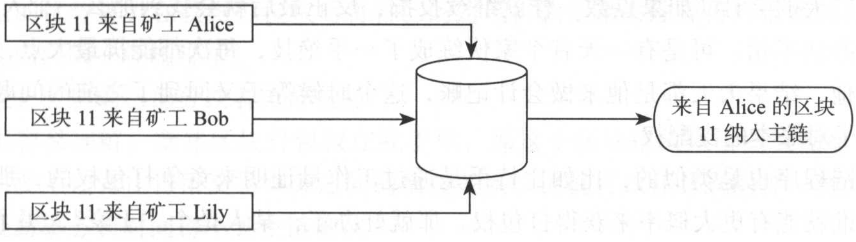
当系统中出现了新版本的软件，并且和之前版本软件不能兼容，老节点无法接受新节点挖出的全部或部分区块，导致同时出现两条链，这种情况称为硬分叉。

无论是软分叉还是硬分叉，对于区块链应用来说都是一件重大的事情，如果新版本在没有取得社区（主要是占据主要算力的矿池用户）一致认可的情况下就强制推行升级，很

有可能就会导致严重的分叉问题，分叉后会发生些什么是很难预料的，目前比特币就出现了数个不同的版本，包括 BitcoinCore，还有新推出的 BitcoinClassic、BitcoinXT 以及 BitcoinUnlimited（无事务块大小和费用限制）。而以太坊在经历了著名的 TheDAO 合约漏洞攻击事件后直接就进行了硬分叉，分为了以太坊经典（ETC）和以太坊（ETH），这其中又纠结了各种价值观问题、利益问题等。

那么，除了上述的版本升级会导致分叉问题，还有别的什么场景吗？假设版本都是统一的，还会有分叉产生吗？当然有，大家要知道，区块链应用程序是没有一个固定的服务器来作为数据的一致性保证的，它靠的是网络共识算法，在异步网络环境下（我们的互联网就是属于异步网络环境），任何一个节点都是独立工作的，它可能会被关机，可能处于网络不良好的环境，而在接受其他节点发过来的区块数据时，也有可能会收到多个临时版本，需要裁决到底使用哪一个，等等。所有节点都只能进行“最终一致性”，最终一致性就是现在还不一致，但是过段时间大家就会依靠规则互相同步达成一致！

在这些情况下导致的分叉属于临时性分叉。这里解释一种情况，在一个节点接收其他节点发送过来的区块数据时，假设当前区块号是 10 号，下一个是 11 号区块。以比特币为例，要等待矿工发送 11 号区块出来，而此时，可能会有多个矿工都挖矿成功，也就是说会发送多个 11 号区块过来，这个时候节点对于接收到的多个区块会都存储下来，等待以后的筛选，最终会淘汰掉其他只剩下其中一个纳入主链接收（网络会以最终最长的那条链为准，这也是为什么在比特币中，会建议交易至少等待 6 个区块确认后才算是确定了），在没有决定哪个区块会进入到最长的那条链时，就会临时性产生分叉，如下图所示：



事实上，这种情况的分叉仅仅是一种竞争过程的中间产物。

接下来让我们来设想另外一种情况。假如某个人，比如 Bob，他组了一个大型的局域网，这个局域网很大，横跨了大江南北，同时这个局域网不与外网相通。现在在这个局域网中安装了大量的比特币节点程序，这些节点程序可以正常挖矿、验证、交易等，但数据就是不能与外网相通。这种情况下，如果过了很长时间，某一天突然让这个局域网能够与外网相通了，能够发现到外网的其他比特币节点了，会发生什么？当这个局域网封闭足够长的时间后再与外网相通，网内和网外的节点还能正常同步吗？实际上这种情况会对比特币的主链网络产生比较大的影响。假如把这个封锁与外网连接的局域网作为一种攻击手段，实际上就是一种针对比特币的“分割攻击”。将一批比特币节点与主网络分割出来的攻击行为称为分割攻击，如果只是延迟一下新区块的广播，则称为“延迟攻击”，这些情况导致

的问题并不是分叉那么简单了，而是会引起其他问题，比如重复支付或者浪费大量矿工的资源。

当然，这些问题并不只是比特币会有，只要是区块链应用，都会面临这些问题。

9.2 达摩克利斯剑：51% 攻击

我们知道，区块链应用是一个点对点的网络程序，彼此之间通过一个共识规则来进行数据的一致性同步，这个共识规则在软件中也就是一个共识算法，比如比特币中的工作量证明（Proof of Work），以太坊中也是使用工作量证明（根据开发计划，以太坊会更改共识算法），还有一些应用会使用其他共识算法，比如 PoS、DPoS 等。这些算法各自也有很多变种，无论是哪一类，其目的都是一致的，就是提供一个相对公平也容易遵守的机制来确保节点区块链数据的一致，谁来运行这些算法程序，那谁就是矿工，也就是运行挖矿程序的节点。矿工通过完成某种证明算法，得到区块数据的打包权，可以将网络中已经发起但还没有打包到主链的事务数据打包存储到新的区块，并且广播给其他节点。倘若某个人通过某种手段，十之八九的打包机会都被他占有的话，那会发生什么？

我们来举个例子，在一个村子里，大家都要干活，干完活后就到会计那去登记一下，记下来今天谁谁谁干了什么，第二天根据劳动量来分配奖金。这种情况下，这个会计就有了很大的权力，于是大家决定不能总是指派某一个人记账，要经常更换，有人就提出了一个办法，第二天谁来记账，通过掷骰子来决定，每个人掷骰子 6 次，加起来的点数谁最多谁就做第二天的会计（如果点数一样就继续投掷，反正最后就会找到那么一个人），大家都觉得这个办法不错。可是有一天有个家伙练成了一手绝技，每次都能掷最大点，大家谁也竞争不过他，结果天天都是他来做会计记账，这个时候等于又回到了之前的问题原点，这个人拥有了对账本的支配权。

区块链程序也是类似的，比如比特币是通过工作量证明来竞争打包权的，那就是说谁的算力大谁就能有更大概率来获得打包权。那就好办了，某人是个“土豪”，买了很多性能最顶配的矿机（专用挖矿的设备），那他是不是在某种程度上能控制区块链的记账了？事实上就是这样的。目前比特币的挖矿算力主要集中在几个矿池，普通计算机能挖到矿的概率已经很渺茫了。当掌握了某个区块链网络中绝对力量的打包权时，就拥有了一定的破坏能力，这种破坏攻击就称为 51% 攻击，为什么叫 51% 而不是 60% 或者 99% 呢，这只不过是个象征性的说法罢了，不用较真。51% 就表明占据了百分百算力的一大半。

话又说过来了，这占据了优势算力，具体怎么个攻击法呢？比特币中的区块是一个个衔接对应的，而其中的交易事务也是通过输入输出的形式一一对应的，就算是随便修改了一个区块的交易事务，可是要想记入到主链中去还得将区块发送出去，等待被其他节点验证后才行，随便做了一个破坏性的修改，根本就不能得到其他节点的验证通过，那不等于没法攻击吗？ OK，我们就来演示一下这个 51% 攻击是怎么一个场景，我们就以使用工作

量证明机制的比特币来说明。

我们来设想一下，当打包权掌握在自己手里后，能干点什么？

（1）修改自己的交易记录，从而实现双花

我们来看一个例子，看看如何通过这种攻击从交易所获得利益。

1）将自己现有的比特币充值到某个交易所（这是为了兑换法币）。

2）自己计算出一个区块链，包含了一条交易信息，比如发送比特币到自己地址中。

3）假设这个自己计算出来的区块链的长度为 10，此时先不向网络广播这个新的区块，而是先到交易所平台将自己现有的比特币换成法币（如人民币）提取出来，这个提取比特币的交易事务会记录在正常的区块链中。

4）假设当提取人民币时，正常的区块链主链的长度还是 9，而我构造的区块链的长度已经是 10 了，此时向网络广播出去，网络会确认我的区块链是正确的，并且会记入到主链中去。

5）此时人民币已经被我提出来了，而我广播出去的第 10 号区块（最新的区块）中并没有包含我向交易所充值的记录，等于比特币还在我的地址中。

至此，一次攻击就完成了，上述只是一个与交易所之间的例子，其实还有很多其他生活中的场景，比如某咖啡店支持用比特币购买咖啡，我支付给咖啡店一定数额的比特币，此时由于是小额支付，咖啡店并不会等到若干个区块确认后才给我做咖啡（按照比特币 10 分钟一个区块的进度，要等上若干个区块认证……算了，还是不喝了），这个时候这笔交易还没有记入到主链中去，严格说还处于内存池，等待被打包，此时我喝着买来的咖啡，然后通过自己的优势算力在获得打包权后将自己的这笔付款记录去除掉再广播新的区块出去，这样我就能白白喝一杯咖啡了。

（2）阻止区块确认部分或者全部交易

这个很容易理解，既然区块打包权在我手里，那这个区块里放入哪些交易事务就是我说了算，只要不违背比特币交易事务之间的衔接关系，也就可以任意剔除掉一些交易事务，使得有些交易事务长时间得不到主链确认。

（3）阻止其他矿工开采到区块

当算力优势非常明显的时候，阻止他人挖矿也就是显而易见能做到的了，这种情况下还会导致其他矿工失去挖矿积极性，导致最后挖矿算力更加集中在少数优势算力的矿工手中。

为了防止算力资源过于集中从而导致一个去中心自治的区块链系统形式上变成了一个中心化系统，人们也一直在寻求更好的共识算法，比如 PoS 算法、DPoS 算法等。这些算法不依赖于算力证明，而是通过持有数字代币的股权随机分配或随机投票选举代表等方法达到效果。迄今为止，各类共识算法都各有优劣，有些区块链系统将共识模块开发成了一个可装配的组件，可以根据需要随时替换新的共识机制，不再像比特币这样写死在代码中。

当然，即便是一段时间内，算力集中在某些矿工手中，也不见得就会马上受到攻击，

毕竟矿工挖矿是要付出代价的（电费、设备损耗等），而占据算力优势的矿工投入更是不菲，花了这么多代价，反过来再攻击网络，使大家不再信任，从而自己挖出的币也就不值钱了，相信矿工们还是会三思的。

最后提醒一下，51% 攻击虽然可以占据打包权，可以决定打包区块中的交易事务，但并不是可以无限制地修改，至少有如下操作通过 51% 攻击是实现不了的：

1）修改他人的交易记录（没有他人的密钥）

2）凭空产生比特币（其他节点不会通过确认，达不成网络共识）

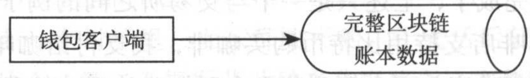
3）改变每区块的比特币发行数量（其他节点不会通过确认，达不成网络共识）

4）把不属于自己的比特币发给别人或自己（除非破解密码）

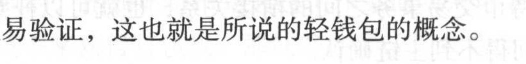
5）修改历史区块数据（其他节点不会通过确认，达不成网络共识）

9.3 简单的代价：轻钱包的易攻击性

我们先来看一下，通常一个标准的钱包应用是什么组成。



钱包之于区块链应用程序来说，是一个前端工具，其作用主要是提供给用户一个交互操作的应用，以便于用户可以通过钱包来进行密钥管理、转账交易、余额查询、合约部署等一系列操作。通过上图我们看到，标准情况下，钱包客户端是与完整区块链账本数据在一起的，对于这些保有完整的、最新的区块链拷贝的钱包应用，称为“完全钱包”，能够独立自主地校验所有交易事务，而不需借由其他的节点服务，除了这种完全钱包，另外还有一种钱包只保留了区块链的一部分，准确地说是只保留了区块头而去除了区块体中的详细事务数据，因此可以大大减少需要同步的数据量，它们通过一种名为“简易支付验证”（SPV）的方式来完成交易验证，这也就是所说的轻钱包的概念。



我们知道，当通过钱包进行一次转账交易时，需要经过一个支付验证，再经过一个交易验证才能有效。支付验证比较简单，主要完成两件事：判断用于“支付”的那笔交易是否已经被验证过以及得到了多少个区块的确认。交易验证就要复杂多了，需要检查余额是否足够，是否存在双花，脚本能否通过等，通常是由运行完全节点的矿工来完成的。

让我们来看一个场景：考虑这样一种情况，Bob 收到来自 Alice 的一个通知，Alice 声称她已经从某某账户中汇款一定数额的钱给了 Bob。去中心方式下，没有任何人能证明 Alice 的可靠性。接到这一通知后，Bob 如何能判断 Alice 所说的是真是假呢？看看下面的验证过程：

1）若是交易验证：Bob 本人想亲自验证这笔交易。首先，Bob 要遍历区块链账本，定位到 Alice 的账户上，这样才能查看 Alice 所给的账户地址上是否曾经有足够的金额；接下来，Bob 要遍历后续的所有账本，看 Alice 是否已经支出了这个账户地址上的钱给别人（是否存在双花欺骗）；然后还要用验证脚本来判断 Alice 是否拥有该账户地址的支配权。这一过程要求 Bob 必须得到完整的区块链才行，也就是说 Bob 节点上必须要有完整的区块链数据副本。

是否有足够金额　是否存在双花　是否有支配权

2）如果 Bob 只想知道这笔支付是否已经得到了验证（验证了就发货），也就是说自己不想做完整的交易验证，只希望通过系统来快速验证两个东西：支付交易是否已经收录于区块链中和得到了多少个区块的确认，这种情况下 Bob 节点是不需要有完整的区块链数据副本的。

交易是否已收录　多少区块确认

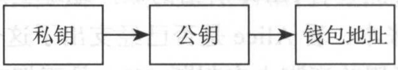
显然，对于日常的支付需求来说，很多时候仅仅做一个支付验证其实就够了，更何况，倘若必须随时随地都要带上一个完整的区块链副本数据，那可是要命了，以比特币来说，目前的数据大小已经有 100 多 GB 了，而且还在不断增长中，怎么可能到处带着，更加不可能在手机这种移动设备上安装了，如此则可用性就太低了。于是 SPV（Simplified Payment Verification，简单支付验证）就应运而生了，就是不运行完全节点也可验证支付的意思，用户只需要保存所有的区块头就可以了，当然由于只保留区块头，因此用户自己是不能验证交易的，需要从区块链某处找到相符的交易，才能得知认可情况。

这种 SPV 钱包也就是轻钱包，大大方便了使用，可是问题也是很显而易见的，由于只是简单通过区块头来验证一下是否存在交易，相当于警卫在进行登记检查的时候，只是看一下牌子号是不是在许可范围内，而不再检查这个人真正的身份来历，这个时候安全性就完全取决于牌子本身的真实性了。事实上 SPV 钱包是把检查的主要工作交给了同事（其他完整节点），同事认真负责就不会有问题，同事如果被掉包了或者叛逆了，那就要出事了。

9.4　忘了保险箱密码：私钥丢失

在日常生活中，能够标识我们身份的那就是身份证了，每个人都有自己的身份证号，我们在银行办了卡如果忘记了密码可以凭身份证去银行重设；我们要去办理住房公积金也是要凭身份证；要买机票火车票等也是要凭身份证。如果身份证丢失了那可就麻烦了，身份证就是我们在这个国家的唯一标识（当然也还有其他证明比如护照、驾驶证，这里暂且不表）。那么，在区块链应用的世界里，唯一标识一个用户的身份的是什么呢？答案就是私

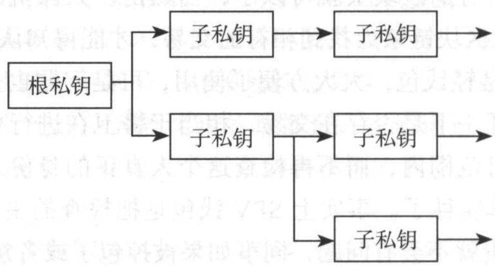
钥。在这里，每个用户都拥有一对密钥：公钥和私钥。



如图所示，用户在办身份证的时候，系统会首先生成一个私钥，然后根据私钥生成公钥，这俩是一对，然后再对公钥进行一些编码处理得到一个钱包地址，几乎所有的区块链应用都是这样一个身份管理过程，只是具体使用的算法不同而已。可以看出，私钥是多么重要，掌握了私钥就什么都掌握了，可以进行转账和任何区块链应用支持的其他操作。

那么，既然私钥如此重要，如果不小心遗忘了该怎么办？凭身份证去哪重设？不好意思，如果私钥丢失了，那就是真的丢失了，没有任何人能够帮你恢复，假如你的某个钱包地址下有大量的数字资产，那可就心塞了。就目前来说，比如比特币系统中，就有很多被遗忘了私钥的地址，其总额加起来价值数十亿美元，可是一点办法都没有。那就不能破解这个密钥吗，既然公钥是公开的，那可以想个办法推导出私钥，大不了慢慢去试。就目前世界范围内广泛使用的公开密钥算法，比如 RSA（基于大素数分解困难度的算法）、ECC（椭圆曲线密码）等，还都没有被破解的先例，如果是去慢慢地试（暴力破解），那在数学概率上是很低的，基本上相当于从唐朝开始试，试到现在也看不见希望的那种。

现在的很多钱包软件，为了追求使用的便捷性，发明了很多钥匙串技术，比如 HD（Hierarchical Deterministic Wallets，分层确定性钱包），使用一个私钥生成更多的私钥，从而使用一把私钥就可以管理自己众多的地址，那可就更不得了了，如果这把私钥丢了，那就是丢了整个钥匙串。

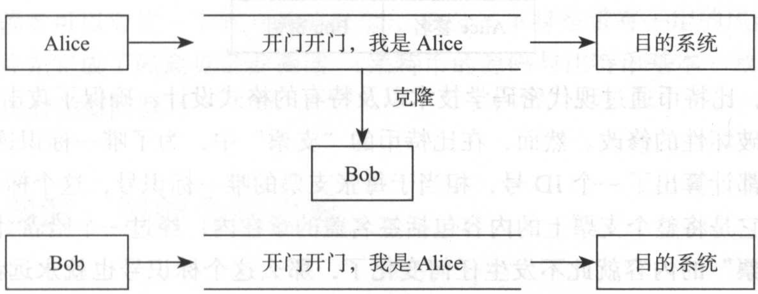


如图所示，方便是方便，不用记住那么多的私钥了，但是方便的反面就是潜在的危险。当然了，这个问题也并非无解，还是可以有一些其他的安全措施的，比如制造一个硬件钱包，跟 U 盘一样往电脑或者手机上一插，通过指纹识别然后启用私钥进行操作；或者通过一段自己能记住的话语来生成私钥，这样能相对有效地防止遗忘。还有一种做法，那就是使用多重签名，这个技术要运用到智能合约，比如 Alice 创建了一份资产合约，其中有价值 1000 的数字货币，此时 Alice 在合约中做了一个规则，当要一下子提出所有的资金时，必须要他本人和 Bob 同时签名（多重签名技术）才行，如果只有一个人签名，则每天只能提取 10 元，此时如果 Alice 忘记了自己的私钥，则可以通过 Bob 慢慢转出这笔钱。

9.5　重放攻击：交易延展性

先解释下什么叫重放攻击（Replay Attack），顾名思义，重放就是重复播放的意思，因此又称为重播攻击或回放攻击，具体是指攻击者发送一个目的主机已接收过的数据包，来达到欺骗系统的目的。我们来举个例子。

Alice 家里安装了一个语音识别的安全门，每次回家的时候 Alice 只要对着门口说一句："开门开门，我是 Alice。"这样门就会打开，自从安装了这样一个门以后，Bob 再也没法偷偷拿 Alice 的钥匙去她家了（根本就用不着钥匙啊）。这可怎么办呢？于是 Bob 偷偷躲在了 Alice 门口的角落里，等 Alice 回家时，用录音笔录下了 Alice 的语音口令，等下次 Bob 再到 Alice 家的时候，就播放这段语音口令，门就能打开了（欺骗了门的识别系统，系统误以为是 Alice 的语音口令）。这就是重放攻击的意思了，合法的主人是使用什么样的通信口令来进行身份认证的，攻击者截获这段通信口令，然后原样的发送给系统，从而欺骗了系统的验证。



那么什么叫交易延展性呢（Transaction Malleability）？延展性是一个形象的称呼。我们知道，在自然界中，有些材料可以经过各种拉伸、锻造来改变形状，但是不会改变材质和质量，比如黄金白银，历史上我们都曾经使用过这两者作为货币，无论是整块的黄金白银还是碎银子碎金子，无论是打造成元宝的形状还是砖头的模样，都不影响它本身的材质和质量，我们只要称一下重量，只要符合重量需求，就能照常花出去（接收者都能验证通过）。区块链应用的转账交易功能，也有这样的现象（当然，对于已经修改了这个问题的应用就不再有这样的问题了），我们以比特币为例来说明这个问题是怎么发生的。

要理解这个问题怎么来的，需要先了解比特币的交易事务的结构，简单地说，比特币在进行转账交易时，会构造一条交易数据，这条数据中包含了转账者的签名、接收者的地址等重要信息，就如同一张支票一般，按照格式填好了信息后，签上名字就发出去了，发到哪？发到比特币网络中，让其他节点来共同见证这笔转账，只要验证没有问题，就会被矿工打包到新的区块中，这就算是转账完成了。可是，大家有没有注意到，就这么一张"支票"发送到网络中，就不怕别人篡改吗？万一某个节点获取到这张"支票"后把金额改掉或者把转账地址改成自己怎么办？放心，这些信息还真改不了。假设 Alice 转账一笔 1000 的金额给 Bob，我接收到这个发出来的"支票"数据了，此时我想进行如下修改：

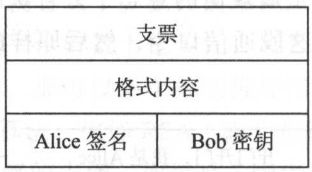
1）将 1000 的金额改成 800；

2）要想修改金额，得拥有 Alice 的密钥，因为这部分的信息是 Alice 用自己特有的密码签名敲章的，没有密钥是没法修改的，而且这个密钥可不像日常生活中的公章那样可以随意冒充，这是由特有的密码算法决定的；

3）将 Bob 的地址替换成我自己的；

4）要想修改 Bob 的地址，得首先解密这一块的信息，因为这一块的数据是使用 Bob 的密钥加密的，只有 Bob 才能解开。

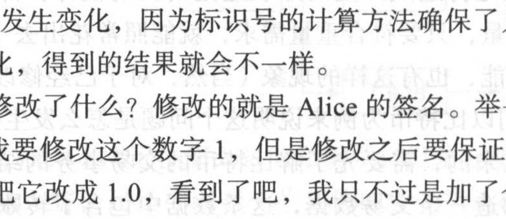
真是一筹莫展啊，那我只想破坏，随便修改掉一些信息行不行？行是行，可是起不到破坏的作用，因为其他节点一旦接收到被修改过的明显有问题的“支票”会直接验证不通过，就被扼杀在襁褓里了。让我们来看一下这个简单的示意图：



如图所示，比特币通过现代密码学技术以及特有的格式设计，确保了攻击者难以对“支票”本身进行破坏性的修改。然而，在比特币的“支票”中，为了唯一标识这么一笔交易，针对每张支票都计算出了一个 ID 号，相当于每张支票的唯一标识号，这个标识号是怎么计算出来的呢？它是将整个支票上的内容包括签名盖的章在内，经过一个哈希计算得出来的，如果这张“支票”的内容就此不发生任何变化了，那么这个标识号也就永远都是那么一个号了，直到被记录到主链区块中去就算是定案了。

现在还记得前面说的延展性吗？是的，如果有一个办法，稍微改变一下支票的某个能修改的信息，但是仍然保证这张支票是有效的，能够通过网络中的节点验证，那会发生什么？那就会导致支票上的标识号发生变化（相当于黄金的形状变化了，但是材质和质量仍然没变），标识号为什么会发生变化，因为标识号的计算方法确保了只要这张“支票”中任何参与计算的内容发生变化，得到的结果就会不一样。

那么，延展性攻击修改了什么？修改的就是 Alice 的签名。举一个容易理解的例子，假如有一个数字 1，现在我要修改这个数字 1，但是修改之后要保证它仍然是代表 1，那怎么修改，很简单，我可以把它改成 1.0，看到了吧，我只不过是加了个小数位而已，这并不能改变这个数字的数学意义，它还是代表 1，可对于标识号的计算方法来说，这就算是内容发生变化了，它就会计算出另外一个支票标识号。



图中的事务 ID 就是"支票"标识号的意思，那么，攻击者通过这样的更改能干嘛呢？它可能会导致以下的后果：

1）接收方无法通过原始的事务 ID 来查询这笔转账；

2）被修改过的交易会与其余在网络中传播的原始交易争抢进入区块，一旦抢先进入了新的区块，原始交易就会被网络中的节点拒绝，虽然不影响转账本身，但是会带来迷惑，而攻击者利用这种迷惑可以达到一些欺骗的目的；

3）阻止原始的交易进入区块。

这种类型的攻击就是属于事务延展性重放攻击。

这个问题有没有解决的方法呢？还是有一些的。其中一项就是隔离见证。隔离见证的方案很简单，既然是因为签名被更改导致的问题，那就将签名从交易数据中分离出来，放到别的地方，这样做还有一个好处，那就是将签名数据从交易数据中分离后，相当于节约了存储空间，等同于扩容了，扩容后就能让一个区块容纳更多的交易记录。当然，这种方法也是很有争议的，比特币社区一直都没有统一意见，其中一个原因就是这实际上是一种软分叉方案（读者可以对比一下软分叉的概念），软分叉本身是带有一定的风险的。2017 年 5 月，莱特币首先完成了隔离见证的激活。（莱特币的源码与比特币基本一致，只是共识算法不一样，因此有类似问题。）

9.6 代码漏洞：智能合约之殇

9.6.1 说说 TheDAO 事件

提起 TheDAO 就不能不先说说以太坊，因为这个事件就是以太坊发展过程中发生的一个重大的安全事件。事实上这个事件到最后已经演变为两种价值观之争，而不再只是技术方面的争论了，以太坊也因为这个事件硬分叉为两个版本：以太坊经典（ETC）和以太坊（ETH）。

先来说下这个事件吧，以太坊属于区块链的二代技术，与比特币这种一代技术的应用比起来，支持更复杂的脚本编程，不但本身实现了数字货币，而且还可以让开发人员通过使用以太坊支持的脚本语言自定义编写所需功能的智能合约，这是一个相当跨越的创新。通过智能合约的实现，人们可以在以太坊上创建自己的数字货币（没错，你可以在以太坊上创建以你名字命名的数字货币）、众筹合约（类似于一个公开透明的基金账户）、自治管理组织（比如创建一个融资租赁公司，创建一个合作翻译的组织等）。以太坊的这些能力引起了人们极大的兴趣，其中就有人通过这些技术特性创建了一份众筹合约，这便是 TheDAO 事件的开始。

大家这里要注意区分一下 DAO 与 TheDAO 的区别，DAO 是 Decentralized Autonomous Organization 的简称，也就是去中心自治组织或者叫分布式自治组织（两种说法有哲学意味

上的差别，这里就不去展开了)，DAO 是以太坊智能合约支持的一个功能，而 TheDAO 是通过这种技术创建并运行在以太坊上的一个智能合约，这是由德国初创公司 Slock.it 开发建立的，这份众筹合约一度融资众筹达到 1.5 亿美元，每个参与众筹的人向众筹合约投资以太坊（其实以太坊本身支持的数字货币，也可以叫以太币），并且根据出资金额获得相应的 DAO 代币，出资人具有审查项目和投票表决的权利。

然而，以太坊本身虽然是健壮的，跟比特币网络一样，通过一系列的区块链技术确保了安全，但是创建在其上的智能合约却未必如此，比特币为什么没出现过这样的问题，因为比特币本身并不支持复杂的脚本编程，只有功能极其简单受限的堆栈指令，以太坊拓展了脚本的功能，使其成为了功能完备的编程脚本。复杂带来了功能的强大，也带来了更多的危险。TheDAO 合约的源码中存在着一个函数调用的漏洞，使得攻击者可以将 TheDAO 资产池中的以太币非法转移给自己。这个问题被发现后，TheDAO 监护人立即提议社区发送垃圾交易阻塞以太坊网络，减缓 TheDAO 资产被转移出去的速度（这个其实本身就是属于一类问题了）。2016 年 7 月，以太坊官方修改了以太坊的源码，在区块高度 1 920 000 强行把 TheDAO 及其子 DAO 的资金转移到了另外一个合约地址，通过这种方式夺回被攻击者控制的 DAO 合约中的币，但是这样却导致以太坊发生了分叉，从而导致变成了两条链：一条为原始的区块链（ETC)，一条是分叉出来的新的链（ETH)。可能有朋友会奇怪，怎么原始的链还会一直存在呢？是的，这是因为以太坊作为区块链应用，是一个去中心分布式的系统，软件升不升级不是创始人能控制的。事实上，ETC 和 ETH 代表了社区的两种价值观，ETC 一方认为无论资金发生了什么样的问题，这个是已经发生的事实，而区块链应用的精神就是不可篡改，账本已经形成了就是形成了，这是必须坚持的原则，ETH 一方认为这是一种违法行为，一种破坏行为，发生在软件系统上的行为不能违法，不能忽略司法的意义，为了坚持一种信仰而任由破坏者攻击是不合适的。

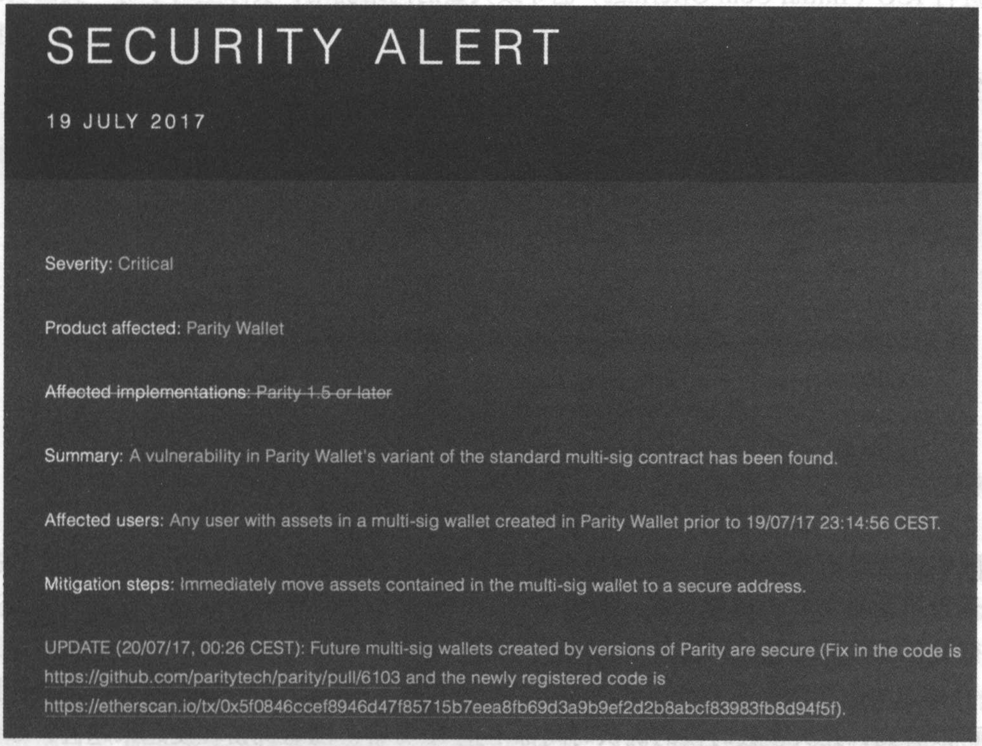
无论如何，这个事件的影响是很大的，也让大家意识到，智能合约还处于发展的初始阶段，区块链应用有很好的创新，很好的技术机制，但当复杂度提高以后，建立在上面的应用也会伴随着各种风险，同时与此相关的各种法律法规建设以及监管制度也亟待探讨建立。未来，相信随着相关的技术标准的逐步建立，代码规范的建立以及底层基础设施的不断进化，很多技术上的问题会一一得以解决。很多时候，破坏性的事件充分暴露潜在的问题，反而会促进技术的进步。

9.6.2 Parity 多重签名漏洞

Parity 是以太坊中使用很受欢迎的一类钱包客户端，它是使用 Rust 语言开发的，这是一种可以用来编写底层系统的开发语言。Parity 在性能上很卓越，运行速度快，系统资源占有少，区块数据的同步也很快。另外，虽然 Parity 是一个全节点钱包，但是同步区块数据的时候对于较早期的区块只保留了区块头而去除了其他数据，因此减少了不少的数据体积，通过 Parity 也能很方便地编写部署智能合约。要详细了解 Parity，可以到这个网址查

看：https://parity.io/parity.html。然而，就在 2017 年 7 月 19 日，发生了一个很严重的 BUG 事故，问题出在 Parity 钱包的多重签名合约库代码。在库代码中存在一个漏洞，使得攻击者可以越权调用合约函数，并将合约中的资产转入到自己的地址，虽然这个漏洞在发现后立即修补了，但是因此而带来的损失和影响却很大。说到这里，大家可能觉得以太坊不是很不安全，又是 TheDAO 事件，又是 Parity 钱包，其实这两者的问题都是出在智能合约的编写上，而不是以太坊本身的问题。如果做个类比的话，以太坊相当于 Windows 这样的操作系统，智能合约则是运行在上面的应用程序。这些事件的发生也告诉我们，当区块链应用支持越来越复杂的功能时，也会放大各种可能的问题概率，大家在编写智能合约时，一定要进行专业的代码审核，任何一个小小的漏洞都有可能导致存入合约中的资产全部丢失。除了要小心合约代码的编写外，站在发展的角度，我们也亟待建立智能合约的编写规范、测试规范等，通过标准化的编写流程来保证安全。

我们看下 Parity 官方发布的漏洞报告：



SECURITY ALERT

19 JULY 2017

Severity: Critical

Product affected: Parity Wallet

~~Affected implementations: Parity 1.5 or later~~

Summary: A vulnerability in Parity Wallet's variant of the standard multi-sig contract has been found.

Affected users: Any user with assets in a multi-sig wallet created in Parity Wallet prior to 19/07/17 23:14:56 CEST.

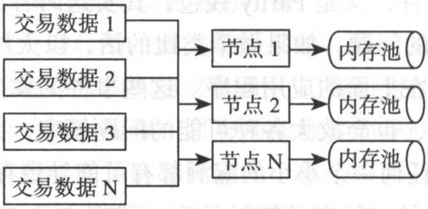
Mitigation steps: Immediately move assets contained in the multi-sig wallet to a secure address.

UPDATE (20/07/17, 00:26 CEST): Future multi-sig wallets created by versions of Parity are secure (Fix in the code is https://github.com/paritytech/parity/pull/6103 and the newly registered code is https://etherscan.io/tx/0x5f0846ccef8946d47f85715b7eea8fb69d3a9b9ef2d2b8abcf83983fb8d94f5f).

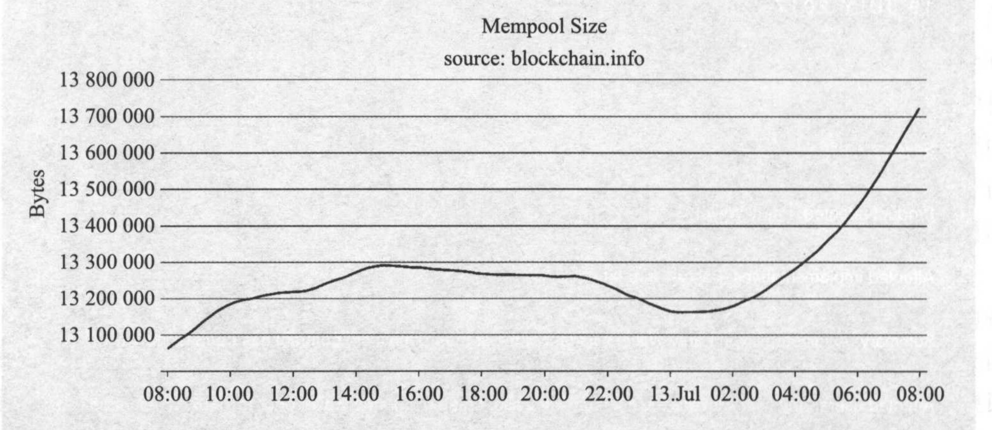
9.7　网络拥堵：大量交易的确认延迟

无论是哪一种区块链应用程序（数字货币、智能合约、去中心的交易系统等），它们的网络都是由一个个独立的节点组成的，发生在节点中的各种操作（如转账交易、合约状态的

变更等)，都会以交易事务的数据形式广播到网络中，通过矿工打包到新的区块，作为主链的一部分而最终确认所有的这些操作。然而，当节点很多，使用量很多的时候，大量发生的交易就会来不及在正常期望的时间内被打包，因为它们都拥堵在网络中，这些等待被确认打包的交易数据通常都维持在节点的内存池中。



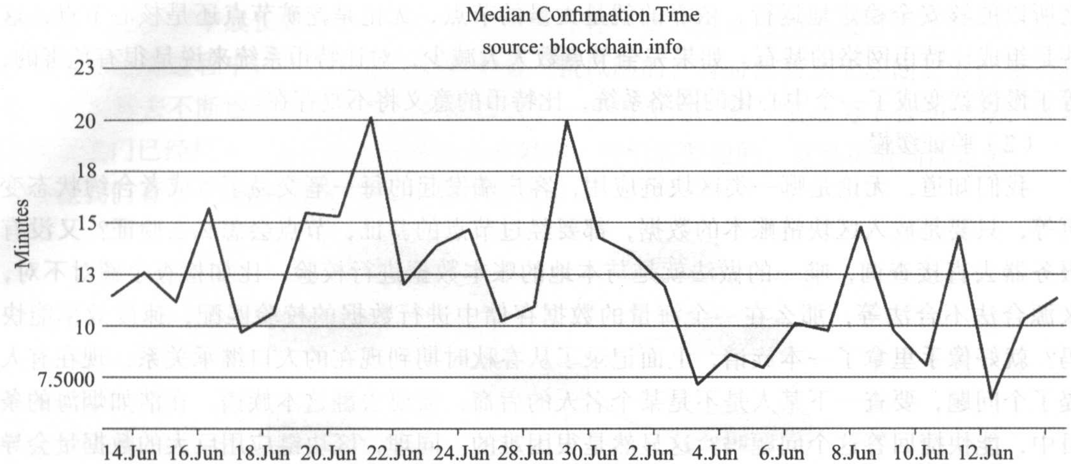
以比特币来说，每隔大约 10 分钟生产一个区块，而每个区块还是有大小限制的。目前来说，比特币一个区块的大小限制是 1MB，而很多人在以太坊上大量地进行智能合约开发以及进行 ICO（Initial Coin Offerings）也导致大量的网络拥堵，实际上对于每一类区块链应用来说，这个问题都是存在的。下图是从 blockchain.info 网站上获取的一段时间内的比特币内存池大小统计：



图中的统计时间区间是：从 2017 年 7 月 12 日的早上 8 点到次日的早上 8 点。数据大小的计量单位是字节，换算一下，平均也有 12MB 多，我们看到，每时每点都充满了这么多等待验证确认的内存池交易数据，比特币平均 10 分钟确认一个区块，一个区块大小还不超过 1MB，可想而知，有多么拥堵了。我们再来看看，实际的交易数据确认时间的统计：

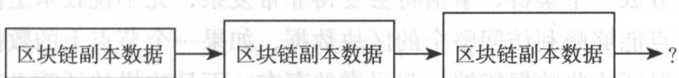
可以看到，在大多数时候的确认时间都不止 10 分钟。随着拥挤程度越来越甚，如果对交易确认速度和区块大小等没有提升的话，将会严重影响比特币的正常使用。其他的区块链应用也是一样，都要面临一旦大规模使用后如何解决网络拥堵的问题。

一般大部分区块链应用还会对内存池中的交易排列优先级进行处理，比如愿意花更多交易费的事务会优先处理等，这将使得使用成本越来越大，对于普通用户的交易，手续费低的就很难排上了。



9.8　容量贪吃蛇：不断增长的区块数据

在说这个问题之前，大家可以试着先去安装一下比特币的核心客户端或者以太坊的客户端，然后进行主网络的数据同步，看看你的硬盘空间能不能承载。在 2016 年 7 月的时候，比特币的区块链账本数据大小才 80GB 左右，而到了 2017 年的 7 月已经是 130GB 之多了。看起来好像问题不大，现在的硬盘动则几百 GB 甚至上 TB（1TB=1024GB），好像还不至于容纳不了这些数据。然而，这里面潜在的问题却并非只是依靠足够的存储容量能解决好的，如下图所示：



如图所示，区块链数据的大小是一直在增长的，对于运行着完全客户端的用户来说，虽然大小的增长可以预先估计，但是这么大量的数据却不那么容易转移。倘若希望在另外一台计算机上运行完全节点，靠慢慢同步那可是有的等了，要直接复制转移那可是上百 GB 的数据，而且大小还一直在增长，无穷无尽，只要比特币一直存在着，数据就会一直在增长，倘若数据量到了 500GB、800GB 乃至上 TB，恐怕到时就连普通硬盘也承载不起了，而以太坊的体积增长更加猛，发展才不过 3 年左右，由于大量的智能合约使用，体积已经超过 200GB 了。那么这样的潜在问题会引起哪些后果呢，仅仅是会占据更多的存储空间吗？

（1）完全节点数减少

巨大的数据量，除了部分用户愿意提供设备外，大部分普通用户是不太愿意让自己的电脑被占据掉那么多的存储空间的，而且这些数据对于用户来说，似乎并没有什么用，日常只是收发转账的话，有一个钱包客户端就足够了。如此，愿意安装完全客户端的用户就会越来越少，这对于比特币网络来说不是一件好事。我们知道，比特币是一个点对点网络，

之所以能够安全稳定地运行，依靠的就是大量的节点，无论是挖矿节点还是核心节点，这些是组成比特币网络的基石。如果完全节点数大大减少，对比特币系统来说是很有危害的，等于慢慢就变成了一个中心化的网络系统，比特币的意义将不复存在。

（2）验证缓慢

我们知道，无论是哪一类区块链应用，客户端发起的每一笔交易事务或者合约状态变更等，只要是放入区块链账本的数据，都要经过节点的验证，节点会怎么去验证？又没有服务器去直接查询，唯一的做法就是与本地的账本数据进行校验，比如检查余额对不对，来源合法不合法等，那么在一个海量的数据存储中进行数据的校验匹配，速度效率能快吗？就好像手里拿了一本族谱，上面记录了从春秋时期到现在的人口继承关系，现在有人提了个问题，要查一下某人是不是某个名人的后裔，需要去翻这本族谱。在浩如烟海的条目中，能快捷回答这个问题吗？这显然是很困难的。同理，区块链应用巨大的数据量会导致数据的验证速度变慢，从而降低了区块链网络的处理效率。

有读者可能会说，像淘宝、京东、微信，它们的数据量也很大，而且恐怕还远远不止一两百 GB，可是并没有发现使用这些软件的功能有多延迟多缓慢啊。这是因为这些系统构造了一整套规模庞大的负载均衡系统，在全国分布有成千上万台服务器，总而言之就是将数据进行了切分，将用户使用的请求分摊到很多的服务器上去。那区块链程序能不能这么干？就目前来说，还是很困难的。首先区块链应用基本都是开源系统，任何人都可以免费下载软件源码，免费运行在自己的设备上，没有任何官方会为此而收费，而构建一个庞大的集群系统，需要大量的设备及人力成本投入；其次，对于区块链应用而言，每个节点必须能够独立运行，尤其是具备完全功能的完全节点，节点之间并没有什么依赖，而如果将一个节点的运行拆分成一个集群，事情将会变得非常复杂，先不说技术上的复杂性，本身也很难保证一个节点能够顺利访问整个的区块数据，如果一个节点上的数据被切分到了多台设备上，那谁能保证这些数据能够一起可靠的存在，而且这样的话就必须保持这些集群服务器是联网的，否则节点将很难独自去验证数据或者访问区块数据，这就增加了不可靠的因素。区块链应用的理念就是通过一个分布式、去中心化的网络结构，通过一套可靠的共识规则实现自治管理系统。如果一个完全节点自身都不能保证总是能可靠地访问完整的区块数据，那还谈何自治管理呢？

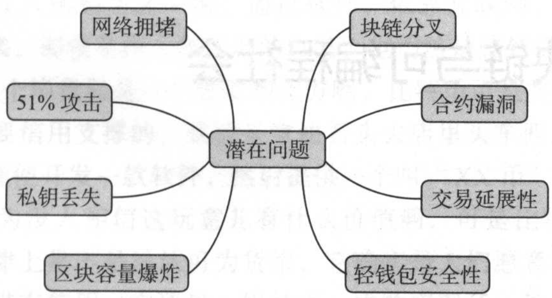
关于这个问题的解决，目前有两个思路：其一就是区块数据的压缩，也就是剔除掉区块链账本中那些已经完全老旧的交易事务；其二就是，同样使用区块链技术来实现一个去中心分布式的自治存储系统。当然，这些都只是一些思路设想，真正要解决问题，还需要做大量的实验论证。

9.9 知识点导图

区块链系统作为一种新型的软件设计技术，在拥有诸多优点的同时，必然也会遇到各

种问题，这是一个成长的过程，我们不必因为存在一些问题而去拒绝或者害怕使用，在人类历史的发展过程中，科技的进步往往不是一帆风顺的，有时候甚至无法断定方向是否正确，而只有去不断地摸索和实验，本章总结了一些潜在的问题，也是为了告知大家，区块链技术之门已经打开，前方会有宝藏也会有陷阱，我们需要做的，就是勇敢向前走去。

让我们看下本章的知识点导图，如下图所示：



Postscript 后记

区块链与可编程社会

1. 未来世界：可编程社会

在漫长的农耕时代，人们建立了各种制度规范，也创造了各种工具，一切似乎都在有条不紊地进行着。由于交通不便，为了与远方的朋友保持联系，人们通过写信来交流；为了方便人们出门在外花费，建立钱庄银号使得可以在异地取款；为了确保生意往来的信用，人们通过字据合同来约定事项，等等。社会大概就是这个样子吧，大家互相配合，各自在自己的角色位置发挥着作用，虽然很多事情还是不那么方便。

不过这些还是会继续发展进化，我们以为跟远方的朋友只能写信交流，后来有了电话，再后来有了互联网；我们以为储存数据只能是一卷一卷的纸张装订，后来有了数据库系统，再后来有了云盘；我们以为跟朋友玩耍只能是逛街吃饭或者唱歌，后来有了网络游戏，再后来有了开心农场；我们以为出门旅行，要么步行，要么马车，后来有了汽车，再后来有了火车和飞机；我们以为……太多了，这发生的一切都在不断改变着我们的生活方式，改变着人们相处的方式，在没有网络之前，能想象可以随时跟远在千里的陌生人聊天吗？能想象买个东西只要在网页上点击吗？能想象随时随地可以了解别的地方发生了什么新闻吗？是的，生产力的发展、科技的进步，大大提高了我们的生活效率，不但如此，也拉进了人与人之间的距离，一切都是那么便捷舒适，生活大概就是这样了吧。

还有什么是没有改变的吗？让我们想一想，物质如此丰富，科技如此发达，我们早已处在文明发达的社会了，可是有一些事，却是千年以来都没怎么被改变过的，比如货币的发行，从原先的黄金白银到后来的纸币，通过中央政府监制发行，这种方式就一直没有实质性改变过；还有商业合同，从古代到现代，也都是立个纸质的字据合同，签上名字盖个章，最多再摁个手印；还有金融买卖交易，在我们通常的思维中，就得有个机构开设一个

平台，然后大家注册登记，再进去交易；再说一个更普遍的，我们人类彼此之间相处了那么长的时间了，一代一代共同在地球上生存了千万年，可是人们之间的信任如何，君不见任何的商业活动，基本都免不了要有个第三方担保吧，这个第三方怎么担保？还得是有个合约，签个字盖个章，等等。这些并没有发生太大的变化。可是，只要社会在发展，终究是会有改变的，直到比特币的出现。

比特币的出现，让人感到为之一振，通过软件，依靠互联网，就那么一组合一捣鼓，突然就能产生货币出来，即便是作为程序员的自己，也仍然是感到很不可思议，古语说“书中自有黄金屋”，可那个毕竟只是一个想象和比方啊，比特币可是实实在在的一个存在，货币是什么？货币是需要信用支撑的，我要是拿块石头去店里买东西，人家肯定会以为我是神经病，我要是自己随便开发一款软件，然后提供一个叫“XX币”的虚拟货币，能用吗？当然是不能用了，因为没人相信这玩意儿有什么价值啊，可是比特币凭什么能让人接受呢？不管比特币在法律上是不是被认可为货币，它确实是人们愿意花钱来购买的东西，也就是说它带有价值，带有信用，它通过一组技术，成功实现了一种通过软件和网络能制造信用和价值的方法，而且这种信用和价值还能够流转。如果你手里有一台机器，它能制造信用，这是什么概念？不单单可以用来发行所谓的数字货币，任何我们人类需要使用到信用的地方都有用武之地，这种信用制造机器是什么？就是我们现今发展的如火如荼的区块链技术。

区块链技术，为我们解决信任和价值传递问题提供了一个新颖而实用的思路方案，我们可以不用只是依靠纸质合约的所谓约定了，将里面的条款编写成智能合约，部署在区块链系统上，系统将会严格地按照事先的约定条件来执行，没有人能够去篡改，也没有人能够撕毁。一切看起来好极了，可是有人说了，那只是能用在商业合约上吗，其他地方还有什么应用吗，不然也谈不上什么虚拟社会呀？正是如此，我们来看一些场景，比如说我们可以在区块链系统上创建公司，公司的每一笔业务都可以永久性记录在区块中，公司与公司之间的合约可以通过智能合约来实现，即便是账务审计，也是相当方便，区块链系统的数据不可篡改性以及随时随地的联网能力，还有先天的公开透明（带许可权限的区块链系统可能需要一个特有的令牌），这一切都使得公司的经济业务运行极其便捷，除了创建公司还能创建什么呢？太多了，可以用来记录历史，想想看，假设原始社会是记录在0号区块的，一直到现在，我们打开这个历史区块链，可以看到各个时期的内容景象，那是多么令人叹为观止啊；还可以用来记录医疗健康信息，再也不用担心找不到本子了，而且医疗机构之间可以共享病历信息，提高对病人的诊断能力，凡此等等，社会的方方面面都会被覆盖到。

随着人们对区块链技术的认识加深，大家终将意识到，与其说这是一类技术，不如说这是一类思想，它代表了一种价值观，公正透明、信任协作的价值观，我们将沿着历史发展的路线，从最初的黄金屋（加密数字货币）走到智能合约，再走向更有前景的可编程社会。

2. 文明的波动：未来已来

文明，这是一个很大的概念，当我们顺着历史的时间轴往回看，可以看到一幅幅波澜壮阔的人类文明发展画面，从曾经那么的野蛮落后，到现代的工业文明，可叹的是我们只能在这滚滚历史长河中见证那么片刻，幸运的是我们正在见证的是一个伟大的时代，从来没有任何一个时代，知识和科技能够以如此爆炸般的速度在发展，如果不是回头看一看，真是在不知不觉中就这么被时代带着走了。看看我们身边现在那些耳熟能详的名词，人工智能、量子通信、虚拟现实、核聚变、区块链等。我们曾经一直都在小说中，在电影中想象着未来的样子，甚至在动画片中描绘着各种未来的场景，所有这些，都表明人们是多么憧憬着更加发达的未来文明。是的，我们总是希望未来快点来，神话可以变成现实，不可能都一个个变成可能，当一个个难题被破解，一扇扇门被打开，我们努力望着远方，试图看清那遥远的缥缈未来，激动时刻的未来，还需要等待多久？

古云“大音希声，大象无形”，我们正在等待那想象中的未来，我们以为还要等待很久很久，可是谁又知道，未来的步伐正在不知不觉快速向我们走来，甚至已经站在了我们的前面，一个科技文明的新起点，已经开启了。不知从何而起，似乎什么都有可能发生，什么都可能被实现，我们习惯相信科技总能够解决我们想解决的一切问题，也总会创造出越来越多的惊叹，文明的发展就像是河流一样，每时每刻都静静地朝着一个方向在流动着，而总会在某一刻，一颗包裹着新科技的石子落进了河流中，激起了新文明的浪花，并且产生了连续的波动，波动所到之处，皆会发生改变，我们就在这一次又一次的波动中，从遥远的蛮荒，赶向文明的未来。

区块链技术就是这其中的一颗石子，就在某年某月某日，它就那么滚落了进来，在它的附近开始激起了一些涟漪，然后有人发现这种技术真是太妙了，它能创造出数字货币，能创建信任网络，它能用来解决太多的问题了，于是产生了越来越多的波动，越来越多的人们开始讨论这种新思想，传统的思路开始发生变革，难以解决的问题开始有了方案，终于它开始爆发了。

让我们做好准备，我们每个人都将会成为新技术文明中的一员，站在风口浪潮之前，迎接这已来到的文明的波动！

作者简介



蒋勇 某集团企业信息技术开发部主管。12 年软件设计开发经历，专注于分布式系统设计，2012 年开始接触比特币底层技术，熟悉比特币、以太坊、超级账本等区块链技术实现，并进行过源码级原理研究。目前从事智能合约安全编码的工具设计开发。



文延（覃文延） 某知名私募投行区块链事业部总经理，是多个自有知识产权的区块链底层技术构架专家委员会和技术白皮书组织者和参与者，北京某央企基金公司区块链研究院副院长。著名 DB2 DBA 和大数据专家，大数据博览会和数博会特邀嘉宾，多次参与国内外国家级数据平台项目研发与管理。在数据库领域深耕多年，曾创立数据库公司 raindb Technologies Inc. 和 rdb.io Inc.，曾在 IBM 多伦多实验室从事 DB2 和 SAP 架构、协议、系统层面研发与技术管理工作。



嘉文 某知名慕课网站架构师，曾长期担任加拿大贝尔在线营销部门与大数据产品部门的经理和技术架构师。加拿大麦吉尔大学信息学硕士、多伦多大学罗特曼商学院 MBA。专注于大数据与分布式数据库系统的分析和研究，从 2012 年开始先后对比特币、以太坊、雷欧币、瑞波、超级账本进行了代码级研究，并开发了基于瑞波的支付网关，基于以太坊的智能合约产品等。目前从事基于区块链的量化交易系统的设计和开发。博客：https://my.oschina.net/gavinzheng731/。

封面设计 · 陈潇然

区块链技术的思想，可以在非信任环境建立信任关系、传递信用与价值，它和具有时代影响力的其他技术一样，盘活了一系列的商业场景，比如防伪、溯源、数据治理、行业链条打通与监控，等等。但是区块链技术的发展尚处于早期，而且其技术栈（分布式系统、共识、加密、分布式账本等）是一种相对紧耦合的状态，技术门槛不低。

针对以上问题，本书致力于降低学习曲线，让更多人了解区块链。本书具有以下特点。

由浅入深

从比特币开始，到区块链技术的骨骼（密码算法）和灵魂（共识算法），再到目前知名的区块链框架介绍，最后从零开始构建一个微型区块链系统，循序渐进。

多图多表

通过流程图与示意图介绍比特币的源码编译、以太坊智能合约的开发部署、超级账本 Fabric 的配置使用、模拟比特币的微型区块链系统的设计实现等，形象而直观。

白话通俗

通过“村民账本记账”“百花村选举记账”等生活化示例，避免多技术组合与新概念上的理解障碍与阅读枯燥感。

上架指导：计算机/区块链

ISBN 978-7-111-58298-4

定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn